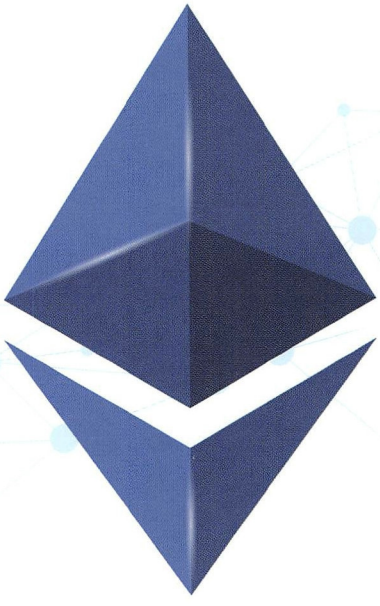
Broadview®
www.broadview.com.cn

“以太坊智能合约 + DApp”从理论到实战

精通以太坊智能合约开发

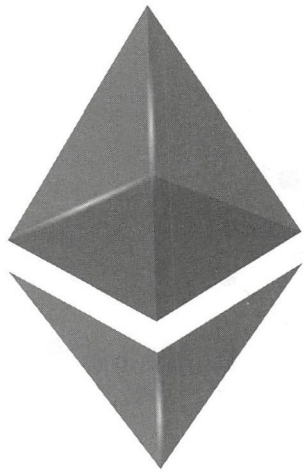
熊丽兵◎编著



中国工信出版集团
電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY
http://www.phei.com.cn

精通以太坊智能合约开发

熊丽兵◎编著



電子工業出版社
Publishing House of Electronics Industry
北京•BEIJING

内 容 简 介

本书系统介绍了以太坊智能合约的开发，对智能合约相关知识进行全面梳理，尤其是对智能合约开发语言 Solidity 进行了详细解读。智能合约的开发者可以从书中获得一些启发和指导。

本书可以作为一本案头手册，方便开发者在开发智能合约时随时查阅。

图书在版编目（CIP）数据

精通以太坊智能合约开发 / 熊丽兵编著. —北京：电子工业出版社，2018.9
ISBN 978-7-121-34951-5

Ⅰ. ①精… Ⅱ. ①熊… Ⅲ. ①分布式数据库－数据库系统 Ⅳ. ①TP311.133.1

中国版本图书馆 CIP 数据核字(2018)第 199195 号

策划编辑：官　杨
责任编辑：牛　勇
印　　刷：三河市华成印务有限公司
装　　订：三河市华成印务有限公司
出版发行：电子工业出版社
　　北京市海淀区万寿路 173 信箱　邮编 100036
开　　本：787×980　1/16　印张：15.75　字数：300 千字
版　　次：2018 年 9 月第 1 版
印　　次：2018 年 9 月第 1 次印刷
定　　价：59.00 元

凡所购买电子工业出版社图书有缺损问题，请向购买书店调换。若书店售缺，请与本社发行部联系，联系及邮购电话：（010）88254888，88258888。

质量投诉请发邮件至 zlts@phei.com.cn，盗版侵权举报请发邮件至 dbqq@phei.com.cn。

本书咨询联系方式：010-51260888-819，faq@phei.com.cn。

前言

本书特色

从 2017 年开始，我在博客《深入浅出区块链》中发表了 30 多篇关于区块链的入门文章，广受大家的好评，也因此收到了电子工业出版社编辑的邀请，希望我写一本关于区块链开发的书籍。我对市面上的书籍做了调查，发现介绍比特币和以太坊入门知识的书比较多，但如果想系统全面地学习智能合约开发，却并没有更好的资源。于是，在跟出版社编辑商议后，我决定写一本全面系统介绍智能合约开发的书，本书由此诞生。

本书并没有对比特币或区块链的基础概念进行过多的介绍，因为市面上已经有很多这方面的文章了，大家也可以参考我的博文《区块链技术学习指引》（https://learnblockchain.cn/2018/01/11/guide/）。本书系统介绍以太坊智能合约的开发，并尽量覆盖智能合约的方方面面，尤其是对智能合约开发语言 Solidity 进行了详细解读。因此本书可以作为一本案头手册，方便开发者在开发智能合约时随时查阅。

本书涉及的 Solidity 内容是以官方文档 0.4.24 版本（https://solidity.readthedocs.io/en/v0.4.24/）为标准的，同时加入了很多我自己的理解以及大量的实例。

另外，本书有时将智能合约简称为“合约”。

读者对象

本书适合那些对区块链有过基本了解，并想进一步学习智能合约或者去中心化应用的开发人员阅读。

本书的读者最好应了解一门语言。例如了解 C、JavaScript、Python 语言会对学习 Solidity 有帮助，因为 Solidity 中的很多思想都参考了这些语言。

本书主要适合以下人员阅读：

- 区块链应用开发者；
- 区块链技术的从业者；
- 对区块链技术感兴趣的人员。

本书内容

第 1 章初探以太坊智能合约，初步认识以太坊、智能合约、Solidity，以及如何使用 Solidity 编写一个简单的智能合约。

第 2 章介绍以太坊核心概念，例如交易、区块、费用 gas、以太坊虚拟机、以太坊账户等概念。

第 3 章介绍一个使用 Solidity 编写的智能合约的组成部分。

第 4 章介绍 Solidity 的类型系统，详细介绍 Solidity 的各种类型，例如整型、布尔类型、地址类型、函数类型、数组类型及结构体类型等。

第 5 章介绍 Solidity 中的单位，包括货币单位和时间单位，通过代码讲解单位如何换算。

第 6 章介绍 Solidity 的全局变量及函数，它们其实就是 Solidity 语言提供的 API，例如获取区块和交易的属性、有关数学及加密功能的函数等。

第 7 章介绍 Solidity 中的表达式、控制结构、变量作用范围、错误处理等。

第 8 章介绍合约，包括如何创建合约、合约函数的可见性、合约函数修饰符等。

第 9 章介绍合约编译、部署、交互、调试，包括对编译器的选择、不同工具的合约部署方法。

第 10 章介绍合约 ABI 的作用，以及如何生成 ABI。

第 11 章介绍编写合约的最佳实践，一是从编码规范来考量；二是从安全性来考量。

第 12 章介绍一些合约案例，如最常见的 ERC20 标准代币合约，介绍如何实现代币增发、资产冻结，以及如何实现一个众筹（ICO）合约，并详细介绍 ERC721 合约的实现方法。

第 13 章介绍去中心化应用的开发，重点介绍如何使用 web3 以合约进行交互，以及 Truffle 框架的使用。

勘误和支持

由于区块链是一种新兴的技术，以太坊平台也处在不断更新发展的过程中，加上作者水平有限，书中难免出现疏漏或错误。如果大家发现问题，请及时反馈给我（可添加我的微信：xlbxiong），我将在图书再版时进行修正，以提供最准确的内容。

以太坊智能合约及 Solidity 最初的内容都是以英文发表的，有个别术语还没有准确的中文翻译，因此我会在括号里注明英文原文。

为了更好地理解，本书部分内容我录制了视频课程，大家可以关注登链学院微信公众号观看学习。

本书的所有代码都被上传到了我的 GitHub（https://github.com/xilibi2003）上，也欢迎大家 Pull Request。

致谢

本书得以面世，离不开很多人的帮助，尤其是各位前辈的指导。

感谢比特币的开创者中本聪，是他带我们进入了数字货币与区块链的世界。

感谢以太坊创始人 V 神（Vitalik Buterin），是他打造了这个开放的智能合约平台，没有这个平台，就不可能有此书。

感谢电子工业出版社的编辑，他们对书稿做了专业、细致、认真的编校工作。

感谢登链科技及登链学院的同事，在我写书的时候他们帮我分担了很多工作。

感谢那些在我学习区块链技术时阅读的博客文章及书籍的作者，虽然我与他们未曾谋面，但我从他们输出的技术内容里获益颇多。

感谢小专栏平台及创始人寂小桦，在我写作博客的时候，在小专栏平台上得到了很多付费用户的认可，这也是我不断写作的动力。

最后要特别感谢我的家人，尤其是我的妻子，在写作本书的这段时间里，我的大女儿不到四岁，小女儿不到一岁，感谢她一直以来对我的支持以及对家庭的付出。

要感谢的人还有很多，难以一一列举，只希望这本书能够为区块链技术在中国的推广和发展做出尽可能多的贡献。

关于作者

熊丽兵，网络 ID：Tiny 熊。

北京航空航天大学硕士，先后加入创新工场及猎豹移动，全面负责数款千万级用户产品的开发及管理工作，2014 年作为技术合伙人参与创建酷吧时代科技。

2016 年投身于区块链技术领域，创立登链科技。CSDN 博客专家，拥有全网访问量最大的区块链技术博客《深入浅出区块链》（learnblockchain.cn），对底层公链技术、区块链技术落地都有较为深入的研究。

我的个人微信号：xlbxiong。欢迎大家向我反馈问题，或者和我一起讨论问题。二维码如下：

我的公众号为：blockchaincore。可以回复 eth_book，获取本书的代码等相关资源。二维码如下：

读者交流 QQ 群：245251041。

目录

第 1 章

初探以太坊智能合约

在本章中，主要带领大家初步认识以太坊、智能合约、Solidity 语言，并介绍如何使用 Solidity 语言编写一个简单的智能合约。

以太坊诞生

自 2008 年比特币出现以来，数字货币逐渐被大家所接受，人们发现其背后的区块链技术在数字货币之外同样有着广阔的应用空间，然而苦于比特币在设计之初仅考虑了数字货币的场景（虽然比特币也支持编程，但它是非图灵完备的，功能有限），因此对于很多商业应用比特币平台无法支持。

2013 年年末，Vitalik Buterin（一位俄罗斯天才少年，人称“V 神”）针对比特币系统非图灵完备性、效率低等缺点，首次提出了以太坊概念，并发布了《以太坊：下一代智能合约和去中心化应用平台》白皮书，启动了项目。

2014 年 7 月，通过 ICO 众筹到了 31529 个比特币，用来支持以太坊项目的发展。

2016 年年初，以太坊的技术得到市场认可，价格开始暴涨，也吸引了大量开发者来到以太坊的世界。

2018 年 5 月，以太币成为市值第二高的加密货币，仅次于比特币，以太坊成为目前热门的区块链应用平台。

智能合约

以太坊上的程序被称为“**智能合约**”，其包含代码和相应的状态数据。以太坊是区块链与智能合约的完美结合，通过编写智能合约可以实现强大的功能，实现去中心化的应用开发。

> 智能合约的英文是 Smart Contract，和人工智能（AI，Artificial Intelligence）的“智能”没有关系。尼克·萨博在 1995 年就提出了智能合约的概念——就是将法律条文写成可执行代码。Vitalik Buterin 把它引入到以太坊中，表示以太坊程序能自动执行及无法被干预的特点。
> 现在智能合约已经扩展到所有的区块链平台，很多时候人们把超级账本、EOS 等区块链平台的程序也称为“智能合约”。

Solidity 语言

以太坊官方推荐的智能合约开发语言为 Solidity，它是一门静态的、支持继承、类库以及复杂的自定义类型等特性的高级语言。Solidity 在设计上借鉴了 Python、JavaScript 等语言，其语法也和 JavaScript 相似。

一个简单的智能合约

我们先看看怎么使用 Solidity 语言来编写一个简单的智能合约。

```
pragma solidity ^0.4.0;

contract SimpleStorage {

    uint storedData;

    function set(uint x) public {
        storedData = x;
    }

    function get() public constant returns (uint) {
        return storedData;
    }
}
```

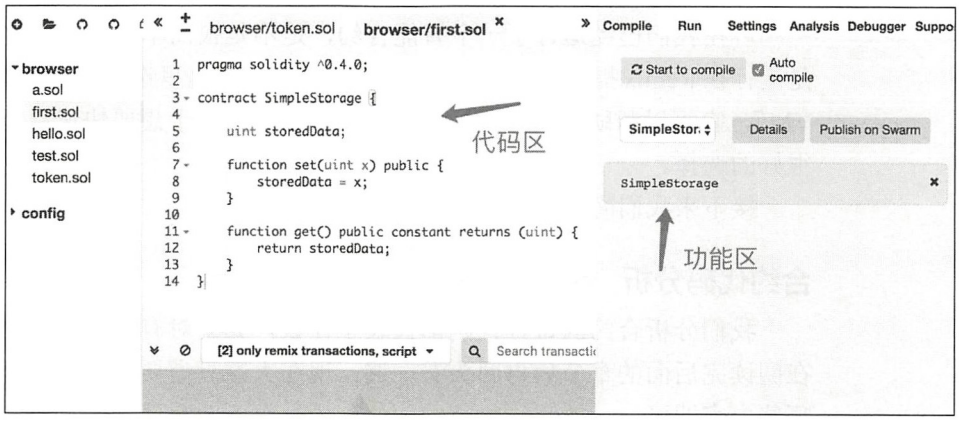
这个智能合约的作用是在区块链上存储一个变量，任何人都可以通过调用set()函数设置改变值（覆盖之前的数字），调用 get()函数获取值，这个数字将会被永久留存在区块链的历史上。当然，这是一个没有多大实际意义的智能合约，这里仅仅是作为一个例子让大家看看智能合约是什么样子的。

> 这个合约就像我们在学习其他语言时，第一个应用是 Hello World 程序一样，之所以不用 Solidity 写一个 Hello World 应用，是因为智能合约运行在分布式网络中，无法像本地应用一样运行输出日志。这也是智能合约不同于传统应用的一个显著区别。
> 另外，用其他语言编写程序时，通常会有一个程序入口方法（如 main 方法），而智能合约没有入口方法，每一个函数都可以被单独调用，并且每一个函数也都只能在合约内部实现，没有实现全局函数。
> 在区块链上存储数据，是智能合约最常见、最基本的功能，这也是我们使用这个例子的原因。在本书后面的章节中，也会多次使用这个例子。

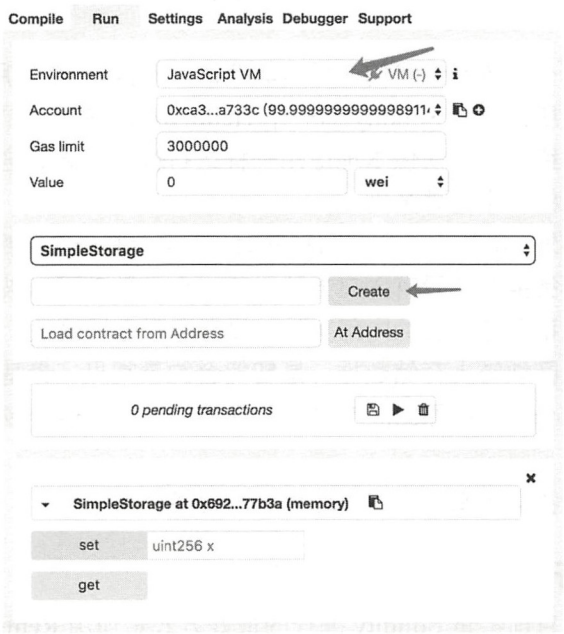
我们再来看看怎么运行这个智能合约。

运行

目前尝试 Solidity 编程的最好方式是使用 Remix。Remix 是一个基于浏览器的 Solidity IDE，它集成了 Solidity 编译器、运行环境，以及调试和发布工具。使用浏览器打开网址 https://remix.ethereum.org/，在所打开的 IDE 界面中的代码区输入上面的代码，如下图所示。



然后在功能区切换到 Run 标签页，在 Environment（环境）一栏选择“JavaScript VM”，点击 Create 按钮（注：新版 Remix 改为 Deploy 按钮了），如下图所示。



至此，第一个智能合约已经创建完成。合约创建完成之后，在功能区的下方会出现智能合约可以调用的所有函数，在这个例子中显示的是 set()和 get()函数，大家可以动手调用一下试试看。

现在我们已经运行了一个智能合约，是不是很简单？不过，这个智能合约是运行在 Remix 提供的模拟区块链环境下的。因为这里选择的环境是“JavaScript VM”，它可以帮助我们准备好账号和以太币，这对于想要快速启动一个合约是很好的选择。

接下来我们简单分析这个智能合约的代码。

合约代码分析

我们分析合约的每行代码都代表了什么，如果对有些内容不理解，则可以在阅读完后面的章节后再回头看一遍，现在大家只需要知道一个合约大概包含哪些内容即可。

```
pragma solidity ^0.4.0;
```

关键字 pragma 用来告诉编译器如何编译这段代码，^表示这里需要 Solidity 0.4.0 及以上版本（在合约版本声明章节中会进一步说明，版本号的第 3 部分可以变，留出来用于修复 bug，如 Solidity 0.4.1 的编译器有 bug，可在 Solidity 0.4.2 中修复，现有合约不用改代码），但是不能高于 Solidity 0.5.0（以避免兼容性问题）。

```
contract SimpleStorage
```

这行代码定义了一个合约，合约的名字为 SimpleStorage（这和在其他语言如 JavaScript 及 Python 中定义一个类很相似，只不过 class 关键字变成了 contract）。一个合约通常由一组代码（合约的函数）和数据（合约的状态）组成。一个合约被存储在以太坊区块链上，对应一个特殊地址。

```
uint storedData
```

这行代码声明了一个变量，在智能合约中称为“状态变量”。这个状态变量名为 storedData，类型为 uint（一个 256 位的无符号整数），可以把它理解为数据库里面的一个存储单元。

```
function set(uint x) public {
    storedData = x;
}
```

上面的代码定义了一个函数 set()。在 Solidity 中通过 function 关键字来定义函数，并且函数的可见性修饰符要写在函数名字及参数之后（这和大部分常见语言不一样），这里 set()函数的作用是修改变量 storedData 的值。

```
function get() public constant returns (uint) {
    return storedData;
}
```

上面的代码定义了一个函数 get()，用来读取变量 storedData 的值。这个函数通过 returns 来指定返回值。returns 返回值要放在所有函数修饰符之后、函数体之前。

> 和大多数语言有点不一样，函数在访问状态变量时，不需要像其他语言那样添加前缀 this。
> 所有的标识符如合约名称、函数名称、变量名称等都只能使用 ASCII 字符集里的字符，但是在字符串变量中可以存储 UTF-8 编码的数据。

一个货币合约的例子

也许通过上一节的例子，你还是不能很好地明白智能合约能做什么，那么现在我们使用智能合约写一个加密货币，这样你就能够体会到智能合约的威力了。在这个例子中会引入很多新的概念，如果你不明白，不用着急。可以学完之后，再回头看看，你会发现使用智能合约来实现一个加密货币是如此的简单。

通过智能合约，我们可以凭空创造出一个货币，而且任何人都可以给其他人发送货币，不需要注册用户名和密码，所需要的只是一个以太坊账户（密钥对）。

代码如下：

```
pragma solidity ^0.4.21;

contract Coin {
    // 关键字 public 让状态变量可以从外部读取
    address public minter;
    mapping (address => uint) public balances;

    // 定义了一个事件，客户端可以根据事件变化做出反应
    event Sent(address from, address to, uint amount);

    // 构造函数，只有在创建合约时运行一次
    constructor()  public {
        minter = msg.sender;
    }

    // 挖矿方法，用来产生新的货币
    function mint(address receiver, uint amount) public {
        if (msg.sender != minter) return;
        balances[receiver] += amount;
    }
```

```
    // 发送货币
    function send(address receiver, uint amount) public {
        if (balances[msg.sender] < amount) return;
        balances[msg.sender] -= amount;
        balances[receiver] += amount;
        emit Sent(msg.sender, receiver, amount);
    }
}
```

这个合约引入了一些新的概念，下面我们详细解读一下。

```
address public minter;
```

这行代码声明了一个可以被公开访问的**地址类型（address）**的状态变量。

地址类型是一个 160 位的值，且不允许任何算术操作。这种类型用来存储合约地址或外部账户地址（本书第 2 章会详细讲账户的概念）。关键字 public 用来表示状态变量的可见性，这一点和其他语言如 Java 和 C++是类似的，不同的是 Solidity 的编译器会自动为 public 状态变量生成一个访问函数（本书第 8 章会进一步讲解），允许在合约之外访问这个状态变量的值。如果没有这个关键字，其他合约是没有办法访问这个变量的。由编译器生成的函数代码大致如下：

```
function minter() returns (address) { return minter; }
```

注意，我们没有必要自己添加这个函数，如果我们添加了一个同名的函数，编译器生成的将不再生效。

```
mapping (address => uint) public balances;
```

这行代码创建了另一个 public 状态变量，它的数据类型是更加复杂的 mapping（在本书第 4 章中会进一步讲解），该类型保存一个个键值对。mapping 可以被看作一个**哈希表**，它会执行虚拟初始化，使所有可能存在的键都对应一个全零的值。不过和其他语言中的 mapping 不一样的是，Solidity 中的 mapping 不能遍历访问（无法获得所有键或值的列表），对于同样的 public 状态变量，编译器会为它生成一个访问函数，函数代码如下：

```
function balances(address _account) public view returns (uint) {
    return balances[_account];
}
```

我们可以通过这个生成的访问函数来查询某个账户的余额。

```
event Sent(address from, address to, uint amount);
```

这行代码声明了一个**事件（Event）**（本书第 8 章会进一步介绍事件概念，第 13 章会进一步讲解如何使用事件），它会在 send 函数的最后一行被发送。应用可以监听区块链上正在发送的事件，一旦事件被发出，监听该事件的 listener 都将收到通知。而所有的事件都包含了 from、to 和 amount 三个参数。例如，可以使用以下代码来监听这个事件。

```
Coin.Sent().watch({}, '', function(error, result) {
    if (!error) {
        console.log("Coin transfer: " + result.args.amount +
            " coins were sent from " + result.args.from +
            " to " + result.args.to + ".");
        console.log("Balances now:\n" +
            "Sender: " + Coin.balances.call(result.args.from) +
            "Receiver: " + Coin.balances.call(result.args.to));
    }
})
```

在监听到该事件被调用时，可以从 result 中获取发送事件所带的参数。

```
constructor () public {
        minter = msg.sender;
    }
```

这是一个构造函数，它只在创建合约时运行。这个构造函数做了一件事，就是保存（外部）函数调用者的地址 msg.sender。

```
function mint(address receiver, uint amount) public {
   if (msg.sender != minter) return;
   balances[receiver] += amount;
}
```

mint 是可以真正被用户或其他合约调用的方法。不过如果 mint 被非合约创建者调用，则什么事也不会发生。

```
function send(address receiver, uint amount) public {
    if (balances[msg.sender] < amount) return;
    balances[msg.sender] -= amount;
    balances[receiver] += amount;
    emit Sent(msg.sender, receiver, amount);
}
```

send 同样是可以真正被用户或其他合约调用的方法。send 可以被任何人（前提是拥有这些币）调用，用来向他人发送币。

我们使用这个合约发送币给一个地址，可能无法在区块链浏览器中查看到相关信息（除非兼容 ECR20 标准，本书第 12 章会详细介绍如何实现 ECR20 代币），因为实际上发送币和更改余额的信息仅仅存储在合约的数据存储器中。不过可以通过使用事件为这个新币创建一个区块链浏览器来追踪交易和余额信息。

本章小结

本章介绍了以太坊如何诞生、何为智能合约，以及两个用 Solidity 语言编写的智能合约示例，让大家对智能合约有了一个初步的认识。

第 2 章

以太坊核心概念

在上一章中，我们大概了解了什么是以太坊及智能合约，本章将介绍以太坊的核心概念。这些概念对于开发智能合约也许不是必须了解的，但是了解这些核心概念，将有助于我们理解以太坊的工作原理，以便写出更好的智能合约。

区块链基础概念

对于程序员来说，区块链这个概念其实不难理解，我们可以先初步理解为分布式数据库，而最难懂的部分比如挖矿、哈希、椭圆曲线加密、点对点网络等，只是为了提供一系列特性和保障。如果不理解，你就把它当成一个特性接受它。相信我，随着知识的积累，你自然而然就理解了。

交易 / 事务（Transaction）

注：英文 Transaction，大多数场合翻译为“交易”，当表示数据库特性时翻译为“事务”。

区块链是全球共享的事务性数据库。**全球共享**意味着参与这个网络的每一个人都可以读取其中的记录。如果想修改这个数据库中的内容，就必须创建一个得到其他所有人确认的事务，**事务**意味着要做的修改（假如想同时修改两个值）只能被完全应用或者一点都没有应用。

> 学习过数据库的读者，应该能够理解“事务”的含义，如果对“事务”一词不是很理解，可以搜索“数据库事务”来学习。

举个例子，想象一张表，里面列出了某个电子货币所有账户的余额。当从一个账户到另一个账户的转账请求发生时，这个数据库的事务特性确保从一个账户中减掉的金额会被加到另一个账户上。如果因为某种原因，往目标账户上增加金额的操作无法进行，那么原账户中的金额也不会发生任何变化。

此外，当一个事务特性被应用到这个数据库的时候，其他事务不能修改该数据库。发送者（创建者）对一个交易进行密码学签名，可以非常直观地理解为该数据库增加了访问保护。在第 1 章的电子货币的例子中，mint()函数中的一个简单检查就可以确保只有持有账户密钥的人才能从该账户向外转账。

区块

一个区块就是若干交易的集合，它会被标记上时间戳和前一个区块的 Hash 标识。区块在经过哈希运算后生成一份工作量证明，从而验证区块中的交易。有效的区块经过全网络的共识之后会被追加到区块链中。

为什么要这样设计呢？因为在比特币及以太坊这样的完全去中心化的区块链中，需要解决被称为**双花攻击**（double-spend attack）的难题，即如果网络中存在两笔相互冲突的交易，它们都想花光同一个账户中的钱时会发生什么情况呢？

实际上网络会自动选择一个交易序列，并打包到区块中，然后在所有参与节点中执行和分发（广播）。如果两笔交易相互冲突，那么最终被确认为后发生的交易将被拒绝，不会被包含到区块中。

这些块按时间形成了一个线性序列，这正是区块链这个词的来源。区块以一定的时间间隔被添加到链上。对于以太坊，这个间隔大约是 17 秒。

作为“顺序选择机制”（也就是**挖矿**）的一部分，可能有时会发生区块被回滚的情况，但通常仅在链的“末端”。在末端增加的块越多，其发生回滚的概率越小。因此交易被回滚甚至从区块链中抹除也是可能的，但交易发生后等待的时间越长，这种情况发生的概率就越小。

> **创世区块（Genesis Block）**是区块链中的第一个区块，其区块序号是 0。它是区块链中唯一一个不指向前一个区块的区块，因为没有前一个区块。只有当网络中的两个节点有相同的创世区块时，它们才会彼此配对。

共识协议：工作量证明（PoW）

以太坊网络中的每一个节点都包含区块链的一个备份。用户想要确保节点不篡改区块链，还需要一种机制检查区块是否合法，如果遇到两个不同的合法区块链（称为**分叉**），则需要确定选择哪一个。

目前以太坊使用工作量证明共识协议防止区块链被篡改。工作量证明系统需要解决一个复杂的数学难题以创建新的区块。解决难题需要大量算力，这就使创建区块很困难。

在工作量证明系统中，创建区块的过程被称为“挖矿”。矿工（Miner）是网络中挖区块的节点。

任何人都可以成为网络中的矿工。每一个矿工都各自解决数学难题，其中第一个解决难题的矿工是胜利者，其得到的回报是 5 个以太币和该区块中全部交易的交易费。如果某个矿工的处理器比网络中的其他节点更强大，则并不意味着这个矿工总会成功，但其成功的概率会比较大。实际上以太坊的网络安全是由网络的全部算力衡量的。

区块链中有多少个区块是没有限制的，可以生成的以太币总数也没有限制。矿工一旦成功挖到区块，就向网络中的所有其他节点广播该区块。区块有一个区块头（Header）和一系列交易。每一个区块都存储前一个区块的哈希值，由此创建一个相连的链。

我们来看矿工是如何解决难题的。矿工首先从所收到的广播中收集新的未挖出的交易，然后过滤掉不合法的交易。合法的交易必须满足正确地使用私钥签名、账户有足够的余额进行交易等条件。现在矿工创建了一个有区块头和内容的区块，其中内容是区块包含的交易列表；区块头包含前一个区块的哈希值、区块序号、随机数（Nonce）、目标值（Target）、时间戳（Timestamp）、难度值（Difficulty）、矿工地址（Address）等内容。时间戳表示区块初始时间。随机数是一个没有意义的值，它是为了设置一个小于或等于目标值的区块哈希。以太坊使用 Ethash 哈希算法，发现随机数的唯一方法是穷尽所有可能。区块头的难度值是目标值的一种不同表述方法。目标值越低，发现随机数需要的时间越多；目标值越高，发现随机数需要的时间越少。

网络中的任何节点都可以检查区块链是否合法，首先检查交易在区块链中是否合法以及时间戳的验证情况，然后检查区块的目标值和随机数是否合法、矿工是否得到合法的回报等。如果网络中的节点接收到两个不同的合法区块链，那么所有区块的整体难度值较高的那个区块链将被视为合法的区块链。

权益证明（PoS）

以太坊最终会切换到使用权益证明，以解决工作量证明过于消耗资源的问题。权益证明的主要思路是：作为验证节点，首先必须拥有一定数量的以太币，根据以太币的数量和时间会产生用于下注验证区块的权益。只有拥有权益的节点才能有效验证区块，当所验证的区块被打包进链后，将获得和权益成正比的区块奖励。如果是验证恶意或错误的区块，那么所下注的权益将被扣除。

以太坊虚拟机（EVM）

在第 1 章中我们介绍过以太坊是一个开放的区块链应用平台，它允许任何人在平台中建立和使用智能合约（以太坊程序）。以太坊虚拟机（EVM，Ethereum Virtual Machine）是以太坊中智能合约的运行环境。

> 熟悉 Java 的读者可以把 EVM 当成 JVM 来理解。EVM 同样是一个程序运行的容器。

以太坊虚拟机是一个被沙箱封装起来、完全隔离的运行环境。运行在 EVM 内部的代码不能接触到网络、文件系统或其他进程，即使是智能合约与其他智能合约也只能进行有限的交互。

然而，以太坊虚拟机本身运行在以太坊节点上，当我们把合约部署到以太坊网络上之后，合约就可以在所有以太坊网络节点的以太坊虚拟机上运行了。

编译合约

在以太坊虚拟机上运行的是合约的字节码形式，需要在部署之前先对合约进行编译，可以选择使用 Remix 或 solc 编译器。在本书第 9 章中会进一步讲解。

账户

这里的账户可以简单理解为银行给我们开设的账户，不过在以太坊中账户不用申请，而是用户根据需要在钱包中生成的。转账行为是由一个账户发起，把财产转移到另一个账户的过程（实际上这个财产也是一个数字）。以太坊也是类似的，同时在以太坊上赋予了账户更多的功能，实际上账户在以太坊中有很重要的作用。以太坊中有两类账户。

- 外部拥有账户（EOA）：该类账户和银行账户很相似，不过它由公钥/私钥对控制，即由人控制，没有关联任何代码。
- 合约账户：该类账户由存储在账户中的代码控制。

外部拥有账户（本书中有时会简称“外部账户”）和合约账户由同样的地址空间来表示。外部账户的地址是由公钥决定的，合约账户的地址是在创建该合约时确定的（这个地址由合约创建者的地址和该地址发出过的交易数量 Nonce 计算得到）。

合约账户存储代码，外部账户则没有。除这点以外，这两类账户对于 EVM 来说是一样的。

但是理解外部账户和合约账户的基本区别依然是很重要的。一个外部账户可以通过创建和用自己的私钥对交易进行签名，以发送消息给另一个外部账户或合约账户。在两个外部账户之间传送的消息只是简单的价值转移（类似于银行账户间转账）。

不过，从外部账户到合约账户的消息会激活合约账户的代码，允许它执行各种动作。比如转移代币、写入内部存储、挖出一个新代币、执行一些运算、创建一个新的合约等操作。

不像外部账户，合约账户不可以自己发起一笔交易。但合约账户可以在响应交易时触发另一笔交易。因此，在以太坊上任何动作都是由外部账户触发的交易所发起的（即动作的发起者必须是外部账户）。

钥匙文件

每个外部账户都是由一对钥匙定义的，即一个私钥和一个公钥。账户以地址为索引（地址就像银行账号的那一串数字），账户地址由公钥衍生而来，取公钥的最后 20 个字节。每对私钥 / 地址都被编码在一个钥匙文件里。钥匙文件是 JSON 文本文件，可以用任何文本编辑器打开和浏览。钥匙文件的关键部分账户私钥，通常用创建账户时设置的密码进行加密。可以在以太坊节点数据目录的 keystore 子目录下找到钥匙文件。因为私钥代表着对账户的所有权，因此我们需要妥善保管好钥匙文件，确保钥匙文件有备份并牢记密码，并尽可能安全地存储它们。

> 如果钥匙文件丢失或忘记密码，就会丢失所有的以太币。没有密码不可能进入账户，也没有“忘记密码”选项。所以一定不要忘记密码。

账户状态

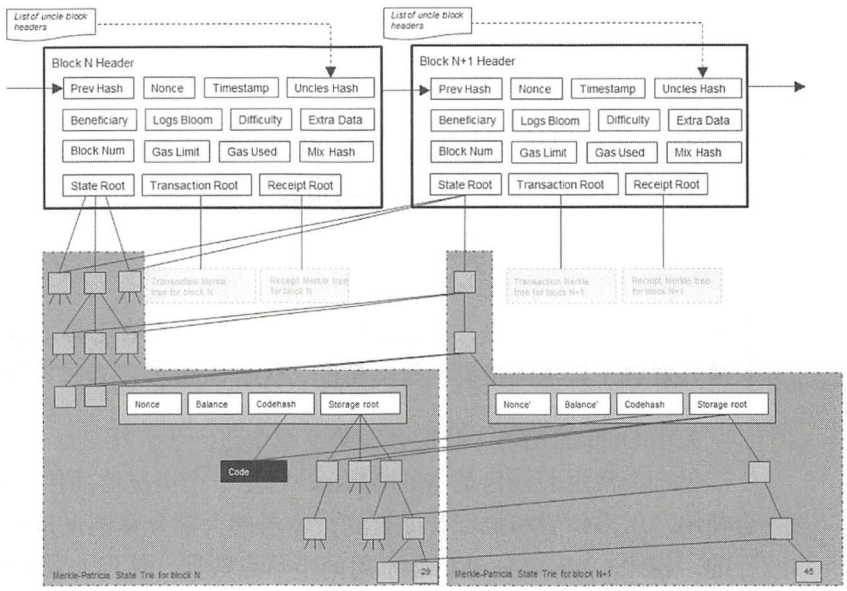
账户状态有四个组成部分，不论账户类型是什么，都存在这四个组成部分。

- Nonce：如果账户是一个外部拥有账户，则 Nonce 代表从此账户地址发送的交易序号；如果账户是一个合约账户，则 Nonce 代表此账户创建的合约序号。

> 注意：以太坊中有两种 Nonce。一种是账户 Nonce，表示一个账户的交易数量；另一种是工作量证明 Nonce，用于计算满足工作量证明的随机数。

- Balance：此地址拥有的以太币余额数量，单位是 Wei，1Ether=10^{18}Wei。当向它发送带有以太币的交易时，Balance 会随之改变。
- storageRoot：Merkle Patricia 树的根节点 Hash 值。Merkle Patricia 树会将此账户存储内容的 Hash 值进行编码，默认是空值。
- codeHash：此账户 EVM 代码的 Hash 值。对于合约账户，就是被 Hash 的代码并作为 codeHash 保存；对于外部拥有账户，codeHash 域是一个空字符串的 Hash 值。

以太坊的全局共享状态是由所有账户状态组成的，它由账户地址和账户状态组成的映射存储在区块的状态树中，如下图所示。



> 以太坊最新的区块总是保存全局共享状态，但每个区块仅仅修改部分状态。

以太坊钱包

以太坊钱包是以太坊账户的管理工具，如使用钱包可以生成账户、账户转账等，不过这只是以太坊钱包最基本的一个功能。实际上在区块链上进行的操作都被当成一笔交易，例如这笔交易可以是转账、挖矿、智能合约的部署和合约函数调用等，全功能的钱包也会提供这些进行交易的功能。

像 imToken 这类在移动平台上使用的以太坊钱包通常只具备最基本的功能。还有一类功能齐全的以太坊钱包，比如 Geth、Mist、Parity 等，在使用这些钱包时会同步整个以太坊区块链的数据，有时也将它们称为以太坊客户端。对于开发者来说，我们可以把它们理解为开发者工具。

Geth 是以太坊官方提供的客户端（钱包），是开发智能合约常用的工具之一，它基于 Go 语言开发。

例如，我们直接在终端输入“geth account new”，就可以创建一个账户。命令如下：

```
$ geth account new
    Your new account is locked with a password. Please give a password. Do not
forget this password.
Passphrase:
    Repeat passphrase:
Address: {0b589e118fa4d38e1e63de5534737e1ee8b944c5}
```

Geth 提供了一个**交互式命令控制台**，在控制台中可以通过接口命令来获取节点信息并和智能合约交互。

> 做过前端开发的人员，应该使用过浏览器的开发者工具。Geth 控制台和浏览器开发者工具里的控制台是类似的，不过 Geth 控制台是在命令行终端运行的。

关于 Geth 的使用，我们会在第 9 章中进一步介绍。

另一类常用的钱包是 MetaMask，它是一个浏览器插件，可以和 Remix 配合使用，用来部署和执行智能合约。本书的大部分案例也是基于 MetaMask 进行的。相比 Geth，MetaMask 不用同步节点数据，使用非常方便。

交易

现在我们结合账户的概念来重新认识一下交易，交易是从一个账户发送到另一个账户（可以是相同的账户或者零账户）的消息。交易可以包含二进制数据负载（Payload）和以太币（Ether）。

如果目标账户包含代码，该代码会执行，Payload 就是输入数据。

如果目标账户是零账户（账户地址是 0），交易将创建一个新合约，新合约地址将由创建者的地址和该地址发出过的交易数量（Nonce）计算得到。这笔交易（创建合约交易）的 Payload 被当作 EVM 字节码执行，输出作为合约代码被永久存储。这意味着为了创建一个合约，不需要向合约发送真正的合约代码，而是发送能够返回真正代码的代码。

一个合约也可以通过一个特殊的指令来创建其他合约（不是简单地向零地址发起调用）。创建合约的调用跟普通的消息调用的区别在于，Payload 数据执行的结果被存储为合约代码，调用者（创建者）在栈上可以得到新合约的地址。

消息调用

合约可以通过消息调用的方式来调用其他合约，或者发送以太币到非合约账户。消息调用和交易非常类似，它们都有一个源、一个目标、数据负载、以太币、gas（费用）和返回数据。事实上每笔交易都可以被认为是一个顶层的消息调用，这个消息调用会依次产生更多的消息调用。

一个合约可以决定剩余 gas 的分配。比如在内部消息调用时使用多少 gas，或者期望保留多少 gas。如果在内部消息调用时发生了费用不足（out-of-gas）异常（或者其他异常），合约将会得到通知（异常会“冒泡”到合约的调用栈）。

当合约调用时，被调用的合约会拥有崭新的内存，以及能够访问调用的 Payload（由被称为“calldata”的独立区域所提供的数据）。当调用执行结束后，返回数据将被存储在调用者预先分配好的一块内存中。

调用层数被限制为 1024，因此对于更加复杂的操作，我们应该使用循环而不是递归。

费用（gas）

无独有偶，以太坊上的每笔交易也会收取一定的费用，这个费用称为“gas”

（有些文章也翻译为“燃料”或“汽油”）。gas 的目的是限制执行交易所需的工作量，同时为执行支付费用。当 EVM 执行交易时，gas 将按照特定规则（这个规则在以太坊黄皮书中有详细说明）被逐渐消耗。

思考费用的作用

施加费用可以防止用户超负荷使用网络。以太坊是一个图灵完备的系统，它允许有循环，并使以太坊受到停机问题的影响，这个问题让你无法确定程序是否无限制地运行。如果没有费用的话，恶意的执行者通过执行一个包含无限循环的交易就可以很容易地让网络瘫痪。因此，费用保护网络不受蓄意攻击。

gas price（gas 价格，以太币计）是由交易创建者设置的，发送者账户需要预付的交易费用（可以理解为给矿工的预算）=gas price × gas limit。如果交易完成还有剩余 gas，这些 gas 将被返还给发送者账户。

无论执行到什么位置，一旦 gas 被耗尽（比如降为负值），就会触发一个费用不足异常，当前调用帧所做的所有状态修改都将被还原。前面介绍过交易（事务）是一个原子操作，要么所有操作全部成功，要么操作失败所有的执行都将被还原，不能出现中间状态。

另外，gas 不仅仅是用来支付计算的费用，也是用来支付存储的费用。

以太坊网络

主网（Mainnet）

以太坊主网或称为“以太坊网络”，是真正产生价值的全球网络，是矿工挖矿的网络。以太坊网络实时数据如区块、散表难度、gas 价格和 gas 花费等，可以在 https://ethstats.net 上查询到。部署在主网上的智能合约，任何应用都可以调用，相关交易信息可以在 https://etherscan.io 上查询到。

测试网络（Testnet）

在主网上任何合约的执行都会消耗真实的以太币，不适合进行开发、调试和测试。因此以太坊专门提供了测试网络，在测试网络中可以很容易获得免费的以太币。测试网络同样是一个全球网络。目前以太坊公开的测试网络有：

- Morden（已退役）。
- Ropsten（https://ropsten.etherscan.io/）。Ropsten 使用的共识机制为 PoW，挖矿难度很低，使用普通笔记本电脑的 CPU 也可以挖出区块。

- Rinkeby（https://rinkeby.etherscan.io）。Rinkeby 使用的共识机制是 PoA（Proof-of-Authority）权威证明。
- Kovan（https://kovan.etherscan.io/）。Kovan 使用的共识机制也是 PoA，目前 Kovan 网络仅被 Parity 钱包支持。

目前开发人员最常用的测试网络是 Ropsten 和 Rinkeby，

使用测试网络依然有一个缺点，即需要花较长时间初始化节点。在实际使用中，测试网络更适合担当如灰度发布之类的角色。

私有网络、开发者模式

我们还可以创建自己的私有网络（通常称为“私有链”）来进行开发、调试和测试。通过上面提到的 Geth 就可以很容易创建一个属于自己的测试网络，在自己的测试网络中，想挖多少以太币就挖多少，也省去了同步网络的耗时。

或者直接使用 Geth 提供的开发者模式。相比私有链，在开发者网络（模式）下，会自动分配一个有大量余额的开发者账户给我们使用。

模拟环境网络

还有一种创建测试网络的方法是使用 Ganache。Ganache 在本地使用内存模拟的一个以太坊环境，对于开发调试来说更方便、快捷。而且 Ganache 可以在启动时帮我们创建 10 个存有资金的测试账户。在进行合约开发时，可以在 Ganache 中测试通过后，再部署到 Geth 节点中。

存储、内存和栈

每个账户都有一块持久化存储区域，被称为**存储（storage）**。其以 key-value 对的形式存储，key 和 value 的长度均为 256 位。在合约中不能遍历账户的存储。存储的读写操作开销较大，修改操作开销更大。一个合约只能对它自己的存储进行读写。

> 开销指的是消耗 gas 的量。

第一个存储区被称为**内存（memory）**。合约在执行每次消息调用时，都有一块崭新的、被清除过的内存。内存是线性的，可以以字节为粒度寻址。但读取的长度被限制为 32 个字节（256 位），写的长度可以是 1 个字节或 32 个字节。

当访问（无论是读还是写）之前从未访问过的内存字（word）时（无论偏移到该字内的任何位置），内存将按字（每个字是 32 个字节）进行扩展。扩容也将消耗一定的 gas。随着内存使用量的增长，其费用也会增高（以平方级别增加）。

第二个存储区被称为**栈（stack）**，EVM 不是基于寄存器，而是基于栈的虚拟机，因此所有的计算都在一个被称为栈的区域执行。栈最大有 1024 个元素，每个元素 256 位。每次只能访问栈顶的元素。限制是通过以下方式进行的：允许复制最顶端的 16 个元素中的一个元素到栈顶，或者交换栈顶元素和下面 16 个元素中的一个。所有其他操作都只能取最顶端的元素（一个或两个或更多元素，取决于具体的操作），运算后把结果压入栈顶。当然也可以把栈上的元素放到存储或内存中。但是无法只访问栈上指定深度的那个元素，除非先从栈顶移除其他元素。

指令集

EVM 的指令集被刻意保持在最小规模，以尽可能避免发生错误而可能导致共识问题。所有的指令都是针对 32 个字节（256 位）这个基本的数据类型操作的。指令集包含常用的算术运算、位运算、逻辑运算、比较运算，以及条件和无条件跳转等。此外，合约可以访问当前区块的相关属性，比如它的序号和时间戳。

委托调用和库

存在一种特殊类型的消息调用，被称为**委托调用（delegatecall）**。它跟上面讲到的消息调用几乎完全一样，不同的是它将目标地址的代码加载到发起调用的合约上下文中来执行，并且 **msg.sender** 和 **msg.value** 不变。例如：A 调用 B，B 委托调用 C，此时 msg.sender 为 A。而如果是普通调用，则 msg.sender 为 B。

这意味着一个合约在运行时可以从另外一个地址动态加载代码。存储当前地址和余额都指向发起调用的合约，只有代码是从被调用地址获取的。这使得 Solidity 可以实现库能力。

日志

以太坊允许日志跟踪各种交易和信息，日志是用一种特殊的可索引的数据

结构来存储的。Solidity 用日志特性来实现事件。创建合约之后就无法访问日志数据了，但是可以从区块链外高效地访问这些日志数据。

由于部分日志数据被存储在布隆过滤器中，因此可以高效并安全地搜索日志。所以那些没有下载整个区块链的网络节点（轻客户端）也可以找到这些日志。

> 布隆过滤器（Bloom Filter）是由布隆（Burton Howard Bloom）在 1970 年提出的。它实际上是由一个很长的二进制向量和一系列随机映射函数组成的，可以用于检索一个元素是否在一个集合中。布隆过滤器的优点是空间效率和查询时间都远远超过一般的算法。

自毁

只有当某个地址上的合约执行**自毁（selfdestruct）**操作时，才会从区块链上移除合约代码。合约地址上剩余的以太币会被发送给指定的目标，然后其（当前和未来的区块上）存储和代码被移除。

注意，即使一个合约的代码不包含自毁指令，也依然可以通过 **delegatecall** 或 **callcode** 来执行这个操作（原文是：Note that even if a contract's code does not contain the *SELFDESTRUCT* opcode, it can still perform that operation using delegatecall or callcode.）。这是 Solidity 官方文档中的一句话，我也没有完全搞明白，不过有一点可以肯定，就是非合约的创建者肯定无法通过这种方式销毁合约。

> 其实合约的删除也依赖以太坊的各种客户端程序实现（可以选择是否删除旧合约）。另外，归档节点可选择无限期保留合约存储和代码。
> 目前，外部账户不能从状态中移除。

以太坊路线图

以太坊的发展分为四个阶段。

前沿（Frontier）

第一阶段，在 2015 年 7 月 30 日发布。

家园（Homestead）

第二阶段，这是以太坊发布的第一个正式版本，在 2016 年 3 月 14 日发布。

大都会（Metropolis）

第三阶段，引入四大特性：zk-Snarks（基于“零知识证明”）、PoS（Proof of Stake，权益证明）早期实施、智能合约更灵活和稳定、使用抽象账户。大都会拆分为两个阶段实施（两个硬分叉）：**拜占庭（Byzantium）**和**君士坦丁堡（Constantinople）**。

- 拜占庭。拜占庭硬分叉在第 437 万个区块高度发生，时间是 2017 年 10 月 16 日，引入了 zk-Snarks 及抽象账户等。
- 君士坦丁堡。预计在 2018 年实施，主要特性就是平滑处理掉所有由于“拜占庭”所引发的问题，并引入 PoW 和 PoS 的混合链模式。

宁静（Serenity）

第四阶段，有两大主要特性：深度抽象和 Casper（基于保证金的权益证明算法）。

本章小结

这一章我们介绍了以太坊的核心概念，包括：以太坊虚拟机、以太坊账户、钱包、以太坊中的交易，以及交易费用（gas）等。有些概念在之后的章节里会多次出现，大家在遇到的时候再回到本章看一看。

第 3 章

Solidity 合约内容

通过前面两章介绍我们初步认识了智能合约及以太坊的核心概念，从这一章开始将逐步介绍用 Solidity 编写智能合约。本章将介绍一个合约通常包含哪些内容。我们将分两个角度来讨论，一是从 Solidity 合约文件结构的角度；二是从合约内容的角度。

Solidity 文件结构

Solidity 合约源文件使用的扩展名为“.sol”。从文件结构上看，一个合约文件通常包含以下几个部分：合约版本声明、引入其他源文件、定义一个合约及注释等。

合约版本声明

Solidity 的源文件需要进行版本声明，告知编译器此源文件所支持的编译器版本，当出现不兼容的新的编译器版本时，它会拒绝编译旧的源文件。经常阅读版本更新日志是一个好习惯，尤其是当大版本发布时。版本声明方式如下：

```
pragma solidity ^0.4.0;
```

这样一个源文件不兼容 Solidity 0.4.0 之前的版本和 Solidity 0.5.0 之后的版本（“^”符号用来控制版本号的第 2 部分）。通常版本号第 3 部分的升级仅仅是

一些小变化（不会有任何兼容性问题），所以通常使用这种方式，而不是指定特定的版本。这样当编译器有 bug 要修复时，不需要更改代码。

如果要使用更复杂的版本声明，那么其声明表达式和 npm 要保持一致，可以参考：https://docs.npmjs.com/misc/semver。

引入其他源文件

Solidity 支持 import 语句，类似于 JavaScript（ES 6），但 Solidity 没有“缺省导出”的概念。

全局引入，引入形式如下：

```
import "filename";
```

从“filename”引入所有的全局符号（包括 filename 从其他文件引入的）到当前的全局作用域。

自定义命名空间引入，引入形式如下：

```
import * as symbolName from "filename";
```

创建一个全局的命名空间 symbolName，成员来自 filename 的全局符号。有一种非 ES 6 兼容的简写语法与其等价：

```
import "filename" as symbolName;
```

分别定义引入，引入形式如下：

```
import {symbol1 as alias, symbol2} from "filename";
```

将创建一个新的全局变量别名 alias 和 symbol2，它们将分别从 filename 引入 symbol1 和 symbol2。

引入路径

引入文件路径时要注意，文件名总是用“/”作为目录分割符，“.”表示当前目录，“..”表示父目录，非“.”开头的路径会被认为是绝对路径。要引用同目录下的文件使用：

```
import "./x" as x
```

而不是：

```
import "x" as x;
```

它会引入一个全局的 include 目录下的 x 文件。

对路径的解析依赖编译器，通常目录层级结构并不一定与本地的文件一一对应，它也有可能是通过 ipfs、http 或 git 建立的一个网络上的虚拟目录。

编译器解析引用文件机制

各编译器都提供了文件前缀映射机制。

- 可以将一个域名下的文件映射到本地，从而让编译器可以从本地加载文件。
- 如果存在多个同名映射，则路径最长的映射最先选用。
- 有一种“fallback-remapping”机制，空串会被映射到“/usr/local/include/solidify”。
- 映射可根据上下文的不同提供对同一实现的不同版本的支持。

对于 **solc 编译器**（一个命令行编译器，在第 9 章会介绍如何安装 solc），它通过命令行参数 context:prefix=target 来提供命名空间映射支持。其中 context: 和 target 是可选的（target 默认为 prefix），所有在 context 目录下的以 prefix 开头的文件都会被替换为 target。

例如，我们克隆 github.com/ethereum/dapp-bin/到本地/usr/local/dapp-bin 目录，并以下列方式使用文件：

```
import "github.com/ethereum/dapp-bin/library/iterable_mapping.sol" as
it_mapping;
```

因此就可以通过以下命令编译这个源文件：

```
solc github.com/ethereum/dapp-bin/=/usr/local/dapp-bin/ source.sol
```

举一个更复杂的例子。如果我们使用一个老版本的 dapp-bin，老版本在 /url/local/dapp-bin_old 下，那么可以使用下列命令编译：

```
solc module1:github.com/ethereum/dapp-bin/=/usr/local/dapp-bin/ \
    module2:github.com/ethereum/dapp-bin/=/usr/local/dapp-bin_old/ \
    source.sol
```

所有在 module2 中应用的文件目录都是老版本，在 module1 中应用的文件目录都是新版本。

注意：solc 仅仅允许从特定的目录下引入文件，它们必须是显式定义的包含目录或子目录的源文件，或者是重映射目标的目录（子目录）。如果想引入直接的绝对路径，那么可以将命名空间重映射为“=/”。

如果有多个重映射指向了同一个文件，那么取最长的那个文件。

如果是 Remix 编译器，则默认会自动映射到 GitHub 上，同样会自动从网络上检索文件，可以使用以下方式引入：

```
import "github.com/ethereum/dapp-bin/library/iterable_mapping.sol" as
it_mapping;
```

定义一个合约

合约的定义和类相似，一个合约需要包含哪些内容将在下一节“合约结构”中详细讲解。

代码注释

代码有两种注释方式，单行注释使用“//”，多行注释使用“/*...*/”。

示例：

```
// this is a single-line comment
/*
this is a
mulit-line comment
*/
```

此外，还有一种**文档注释**（natspec comment），使用“///”或“/**…*/”，通常位于函数或语句的上方。在文档注释中可以使用Doxygen风格的标签（tag）来说明函数作用、参数验证的注解，同时支持文档的生成。

示例：

```
pragma solidity ^0.4.0;

/** @title Shape calculator. */
contract shapeCalculator {
    /** @dev Calculates a rectangle's surface and perimeter.
      * @param w Width of the rectangle.
```

```
     * @param h Height of the rectangle.
     * @return s The calculated surface.
     * @return p The calculated perimeter.
     */
    function rectangle(uint w, uint h) returns (uint s, uint p) {
        s = w * h;
        p = 2 * (w + h);
    }
}
```

合约结构

Solidity 合约的定义和面向对象语言中的类很相似，每个合约都可以包含状态变量、函数、函数修改器、事件、结构类型和枚举类型。另外，合约也支持继承。这一节大家只需要对合约包含哪些内容有一个大致的了解即可，在之后的章节中会详细介绍具体的内容。

状态变量（State Variable）

状态变量和其他语言中类的成员变量很相似，状态变量会被永久存储在合约的存储空间里。

```
pragma solidity ^0.4.0;

contract SimpleStorage {
    uint storedData;   // 这是一个状态变量
    // ...
}
```

函数（Function）

函数是合约中可执行的代码单元。下面是一个函数的例子。

```
pragma solidity ^0.4.0;

contract SimpleAuction {
    function bid() public payable { // 这是一个函数
        // ...
    }
```

```
}
```

关于函数的可见性及访问控制请参考第 8 章。

函数修改器（Function Modifier）

函数修改器可以以声明的方式补充函数的语义。关于函数修改器请参考第 8 章。

```
pragma solidity ^0.4.11;

contract Purchase {
    address public seller;

    modifier onlySeller() { // 声明了一个修改器
        require(msg.sender == seller);
        _;
    }

    function abort() public onlySeller { // 修改器的使用
        // ...
    }
}
```

事件（Event）

事件是以太坊虚拟机（EVM）日志基础设施提供的一个便利接口，用于获取当前发生的事件。

```
pragma solidity ^0.4.0;

contract SimpleAuction {
    event HighestBidIncreased(address bidder, uint amount); // 定义事件

    function bid() public payable {
        // ...
        emit HighestBidIncreased(msg.sender, msg.value); // 触发事件
    }
}
```

在第 8 章中我们将详细讲解事件。

结构类型（Struct Type）

结构类型是一组用户定义的变量组合在一起形成的类型。我们将在第 4 章中详细讲解结构类型。

```
pragma solidity ^0.4.0;

contract Ballot {
    struct Voter {      // 结构类型
        uint weight;
        bool voted;
        address delegate;
        uint vote;
    }
}
```

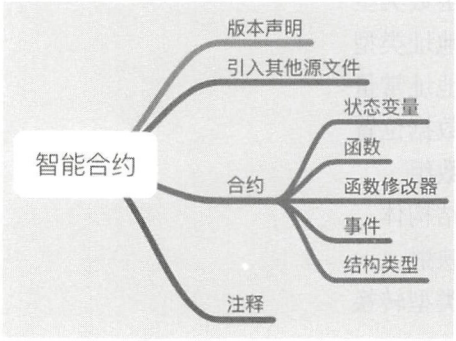
枚举类型（Enum Type）

枚举类型是用户创建的包含几个特定值的集合的自定义类型。

```
pragma solidity ^0.4.0;

contract Purchase {
    enum State { Created, Locked, Inactive } // 枚举类型
}
```

本章小结

一个智能合约的主要结构用图概括如下：



第 4 章

Solidity 数据类型

Solidity 是一种静态类型语言，这一章我们将深入介绍 Solidity 的数据类型。主要内容包括：

- 类型概述及分类
- 布尔类型
- 整型
- 定长浮点型
- 定长字节数组
- 有理数和整型常量
- 字符串常量
- 十六进制常量
- 枚举
- 函数类型
- 地址类型
- 地址常量
- 数据位置
- 数组
- 结构体
- 映射
- 类型转换

- 类型推导
- 运算符

类型概述及分类

Solidity 是一种静态类型语言，常见的静态类型语言有 C、C++、Java 等，静态类型意味着在编译时需要为每个变量（本地或状态变量）都指定类型（或至少可以推导出类型）。

Solidity 数据类型看起来使用很简单，但却是最容易产生漏洞的地方（如出现溢出等问题）。有一点大家也需要关注，就是 Solidity 的类型非常在意所占空间的大小。另外，Solidity 的一些基本类型可以组合成复杂类型。

Solidity 类型分为两类：值类型（Value Type）和引用类型（Reference Type）。

另外，不同类型还可以与不同的运算符组合，支持表达式运算，并按照表达式的执行顺序（Order of Evaluation of Expression）来执行。

值类型

值类型所占空间在 32 个字节以内，值类型变量在赋值或传参时总是进行值拷贝。

值类型包括：

- 布尔类型（Boolean）
- 整型（Integer）
- 定长浮点型（Fixed Point Number）
- 定长字节数组（Fixed-size Byte Array）
- 有理数和整型常量（Rational and Integer Literal）
- 字符串常量（String Literal）
- 十六进制常量（Hexadecimal Literal）
- 枚举（Enum）
- 函数类型（Function Type）
- 地址类型（Addres）
- 地址常量（Address Literal）

引用类型

引用类型主要包括：数组（Array）、结构体（Struct）和映射（Mapping）。

布尔类型（Boolean）

布尔类型使用 bool 关键字声明，声明方式如下：

```
bool isActive;

bool isOk = false;   // 带默认值
```

布尔类型可能的取值为常量值 true 和 false。

布尔类型支持的运算符如下。

- !，逻辑非。
- &&，逻辑与。
- ||，逻辑或。
- ==，等于。
- !=，不等于。

注意：运算符“&&”和“||”是短路运算符，如 f(x)||g(y)，当 f(x)为真时，则不会继续执行 g(y)；f(x) && g(y)，当 f(x)为假时，则不会继续执行 g(y)。

整型（Integer）

和 Java 等语言用 short、int、long 来表示整型有些不一样，Solidity 的整型用 int 加一个类型所占位数的数字一起来表示，这种方式和 Go 语言一致。

整型的关键字有 int8、int16 到 int256，数字以 8 步进。对应的无符号整型有 uint8 到 uint256，uint 和 int 默认对应的是 uint256 和 int256。

声明方式如下：

```
int8 x = -1;
int16 y = 2;
int32 z ;
```

整型支持的运算符如下。

- 比较运算符：<=、<、==、!=、>=、> （返回布尔值 true 或 false）。
- 位运算符：&、|、^（异或）、~（位取反）。
- 算术运算符：+、-、一元运算符“-”、一元运算符“+”、*、/、%（取余数）、**（幂）、<<（左移位）、>>（右移位）。

说明：

- 整数除法总是截断的，但如果运算的是常量（常量稍后讲），则不会截断。
- 整数除 0 会抛异常，即 x/0 为非法的。
- 右移位和除是等价的，如 x >> y 和 x / 2**y 是相等的。左移位和乘等价，如 x << y 和 x * 2**y 是相等的。移位运算的结果的正负取决于运算符左边的数。右移位一个负数，向下取整时会为 0。
- 不能进行负移位，即运算符右边的数不可以为负数，否则会抛出运行时异常，如 3 >>–1 为非法的。

我们来看看下面一个例子。

```
pragma solidity ^0.4.18;

contract testType{

    function add(uint x, uint y) public returns (uint z){
        z = x + y;
    }

    function divide(uint x, uint y ) public returns (uint z) {
        z = x / y;
    }

    function leftshift(int x, uint y) public returns (int z){
        z = x << y;
    }

    function rightshift(int x, uint y) public returns (int z){
        z = x >> y;
    }

}
```

关于代码的运行请参考第 1 章。

运算符简写

和 C、C++、Java 类似，对于一些运算符运算赋值支持以下简写。

a += e 等价于 a = a + e，类似的操作符有：–=、*=、/=、%=、|=、&=、^=。

a++和 a--等价于 a += 1 和 a-= 1，但表达式仍为 a 的值。而--a 和++a 则返回更改之后的值。

整型溢出问题

在使用整型时，要特别注意整型的大小及所能容纳的最大值和最小值，很多合约就是因为溢出问题导致了漏洞，比如美链的漏洞，大家可以参考博客文章（https://learnblockchain.cn/2018/04/25/bec-overflow/）了解。

下面列举几个关于溢出问题的例子。

```
uint8 x = 0x80;
uint8 y = x * 2;   // y 溢出后为 0
uint8 i = 0xf0;
uint8 j = 0x10;
uint8 k =  i +  j;  // k 溢出后为 0
uint8 m = 0x01;
uint8 n =  m - 2;  // n 溢出后为 255
```

避免溢出的一个方法是在运算之后对结果值进行一次检查，比如对上面的 k 做一个检查，如使用 assert(k >= i)。也推荐大家在进行加、减、乘、除运算时使用 OpenZeppelin 的 SafeMath 库，代码的 GitHub 地址为 https://github.com/OpenZeppelin/openzeppelin-solidity/blob/master/contracts/math/SafeMath.sol。

定长浮点型（Fixed Point Number）

定长浮点型的功能和其他语言的浮点型 float 和 double 差不多，都是用来表示浮点数的。但是定长浮点型又有不一样的地方，它需要在声明时指定类型所占的大小，以及小数点的位数，而传统的浮点型 float 和 double 通常值会不一样，即所占用的空间不一样。

Solidity 还不完全支持定长浮点型，它可以用来声明变量，但不可以用来赋值，因此当前还没法使用定长浮点型。

定长浮点型的声明方式如下：

```
ufixed32x1 f;
ufixed32x1 fi = 0.1;   // 错误：UnimplementedFeatureError: Not yet implemented
```

fixed/ufixed 表示有符号和无符号的固定位浮点数。关键字为 ufixedMxN 和

ufixedMxN，其中 M 表示这种类型要占用的位数，以 8 步进，可以为 8～256 位；N 表示小数点的个数，可以为 0～80。

fixed/ufixed 分别代表 fixed128x18 和 ufixed128x18。其支持的运算符如下。

- 比较运算符：<=、<、==、!=、>=、>（返回布尔值 true 或 false）。
- 算术运算符：+、–、一元运算符“–”、一元运算符“+”、*、/、%（取余数）。

注意：它和大多数语言的 float 和 double 不一样，这里的 M 表示整个数占用的固定位数，包含整数部分和小数部分。因此，当用一个小位数（M 较小）来表示一个浮点数时，小数部分几乎会占用整个空间。

定长字节数组（Fixed-size Byte Array）

定长字节数组是指一个所占空间固定的数组，每个元素都是一个字节。由于变长数组不是值类型，因此我们会用单独一节来介绍。

定长字节数组的声明方式如下:

```
byte bt0;
bytes1 bt1 = 0x01;
bytes2 bt2 = "ab";
bytes3 bt3 = "abc";
```

关键字有 bytes1，bytes2，bytes3，…，bytes32，以步长 1 递增，byte 代表 bytes1。

在实际使用中，定长字节数组经常被用来代替字符串（如声明中的 bt2 及 bt3）。

定长字节数组支持的运算符如下。

- 比较运算符：<=、<、==、!=、>=、>（返回布尔值 true 或 falsa）。
- 位运算符：&、|、^（按位异或）、~（按位取反）、<<（左移位）、>>（右移位）。
- 索引（下标）访问：如果 x 是 bytesI，当 0≤ k < I 时，则 x[k]返回第 k 个字节（只读）。

移位运算和整型类似，移位运算结果的正负取决于运算符左边的数，且不能进行负移位。例如可以–5<<1，不可以 5<<–1。

定长字节数组有成员变量：**.length**，表示这个字节数组的长度（不能修改）。

例如获取 bt2 和 bt3 的长度：

```
bt2.length;  // 返回 2
bt3.length;  // 返回 3
```

有理数和整型常量（Rational and Integer Literal）

有理数和整型常量是表达式中直接出现的数字常量，不过也有人把它称为字面量。

整型常量是由一系列 0~9 的数字组成的，以十进制数表示。比如：八进制数是不存在的，前置 0 在 Solidity 中是无效的。

十进制小数常量（Decimal Fraction Literal）带了一个“.”，在“.”的两边至少有一个数字，有效的表示如 1.、.1 和 1.3 等。

它也支持科学符号，基数可以是小数，指数必须是整数，有效的表示如 2e10、-2e10、2e-10、2.5e1 等。

数字常量表达式本身支持任意精度，也就是不会发生运算溢出或除法截断。但是当它被转换成对应的非常量类型，或者将它与非常量进行运算时，就不能保证精度了。比如(2**800 + 1)–2**800 的结果为 1（uint8 整型，尽管中间结果已经超过计算机字长）；.5 * 8 的结果是 4（尽管有非整型数参与了运算）；5/2 + 5/2 的结果是 5（会转为有理数，不会被截断，不过在早期的版本中会被截断）。

只要操作数是整型，整型支持的运算符就适用于整型常量表达式。如果两个操作数都是小数，则不允许进行位运算，指数也不能是小数。

示例代码如下：

```
pragma solidity ^0.4.18;

contract testType2{
    function interLiteral() public returns (uint, uint) {
        return ((2**800 + 1) - 2**800,  0.5*8);
    }
}
```

注意：Solidity 的每一个数字常量都有对应的数字常量类型，整型常量和有理数常量属于数字常量类型。所有的数字常量表达式的结果都是数字常量。

在数字常量表达式中一旦含有非常量表达式，它就会被转换为非常量类型。

不同类型之间没法进行运算，因此下面的代码会编译出错。

```
uint128 a = 1;   // a 不再是常量类型
uint128 b = 2.5 + a + 0.5;
```

上述代码编译不能通过，因为 b 会被编译器认为是小数。

字符串常量（String Literal）

字符串常量是指由单引号或双引号引起来的字符串（如"foo"、'bar'）。并不像 C 语言那样字符串包含结束符，如"foo"这个字符串的大小仅为 3 个字节。和整型常量一样，字符串的长度类型可以是变长的。字符串可以隐式转换为 byte1，…，byte32，如果适合也会转换为 bytes 或 string。

```
"Tiny 熊";
"Tiny\u718A";
```

字符串常量支持转义字符，比如\n、\xNN、\uNNNN。其中\xNN 表示十六进制值，最终会转换为合适的字节数组；而\uNNNN 表示 Unicode 编码值，最终会转换为 UTF8 的序列。

注意：字符串常量不支持任何运算符，比如在其他语言中可以通过“+”来拼接两个字符串常量，但是在 Solidity 中是不可以的。

十六进制常量（Hexadecimal Literal）

十六进制常量以关键字 hex 开头，后面紧跟用单引号或双引号包裹的字符串，内容是十六进制字符串，如 hex"001122ff"，它的值会用二进制数形式来表示。

十六进制常量和字符串常量类似，也可以转换为字节数组。

示例代码如下：

```
pragma solidity ^0.4.18;

contract testType2{

    function hexLiteralBytes() public returns (bytes2, bytes1, bytes1) {
        bytes2 a = hex"aabb";
```


```
        return (a, a[0], a[1]);
    }
}
```

枚举（Enum）

在 Solidity 中枚举可以用来自定义类型。它可以与整数进行显式转换，但不能进行隐式转换。显式转换会在运行时检查数值范围，如果不匹配将会引发异常。枚举类型应至少有一个成员。下面是一个枚举的例子。

```
pragma solidity ^0.4.0;

contract test {
    enum ActionChoices { GoLeft, GoRight, GoStraight, SitStill }
    ActionChoices choice;
    ActionChoices constant defaultChoice = ActionChoices.GoStraight;

    function setGoStraight() {
        choice = ActionChoices.GoStraight;
    }

    function getChoice() returns (ActionChoices) {
        return choice;
    }

    function getDefaultChoice() returns (uint) {
        return uint(defaultChoice);
    }
}
```

函数类型（Function Type）

Solidity 中的函数也可以是一种类型且属于值类型。可以将一个函数赋值给一个函数类型的变量，也可以将一个函数作为参数进行传递，还可以在函数调用中返回一个函数。下面是一个把函数赋值给变量的例子。

```
contract FunctionSelector {
  // 变量 f 可以被赋值为函数 a 或函数 b
  function select(bool useB, uint x) returns (uint z) {
    var f = a;
    if (useB) f = b;
    return f(x);
  }

  function a(uint x) returns (uint z) {
    return x * x;
  }

  function b(uint x) returns (uint z) {
    return 2 * x;
  }
}
```

函数类型有两类：内部（internal）函数和外部（external）函数。

内部函数只能在当前合约内被调用（在当前代码块内，包括内部库函数和继承的函数），它不会创建一个EVM消息调用，访问方式是直接使用函数名f()。

外部函数通过 EVM 消息调用，它由地址和函数方法签名两部分组成，访问方式是this.f()。外部函数可作为外部函数调用的参数或返回值。

函数类型定义如下：

```
function (<parameter types>) {internal|external} [pure|constant|view|payable]
[returns (<return types>)]
```

如果函数不需要返回，则省去 returns (<return types>)，函数类型默认是internal，因此 internal 可以省去。但与此相反，合约中函数本身默认是 public的，仅仅是当作类型名使用时默认才是 internal。

internal 和 external 既表示调用方式也表明了其可见性，很多人在编写传统程序的时候，不是很关注可见性。但写智能合约时大家要特别注意，如果应该使用 internal，却使用了 external，则很容易引发安全问题，同时也增加了 gas的消耗。

注意一下，声明一个函数和声明一个函数类型的变量是不一样的，在第 8章合约的抽象函数中我们会进一步介绍。

有两种方式访问函数，一种是直接用函数名 f，另一种是用 this.f。前者用于内部函数，后者用于外部函数。

如果一个函数变量没有初始化，那么直接调用它将会产生异常。如果 delete 了一个函数后调用，则也会发生同样的异常，例如下面的代码就会发生异常：

```
function select(bool useB, uint x) returns (uint z) {
    var f = a;
    if (useB) f = b;
    delete f;
    return f(x);
  }
```

如果外部函数类型在 Solidity 的上下文环境以外的地方使用，那么他们会被视为 function 类型。它会编码为 20 字节的函数所在地址，和在它之前的 4 字节的函数方法签名一起作为 bytes24 类型。合约中的 public 的函数，可以使用 internal 和 external 两种方式来调用。internal 访问形式为 f, external 访问形式为 this.f。

> 函数类型及下面介绍的数组等类型和其他语言中的类有点像，这些类型还有成员变量（在 Solidity 中称为“属性”）和成员函数。

selector 成员属性

公有或外部（public /external）函数类型有一个特殊的成员属性 selector，它对应一个 ABI 函数选择器（ABI 函数选择器在第 10 章会有进一步的介绍，这里只要知道它是一个函数签名即可）。

```
pragma solidity ^0.4.16;

contract Selector {
function f() public view returns (bytes4) {
    return this.f.selector;
}
}
```

下面的代码显示内部（internal）函数类型的使用：

```
pragma solidity ^0.4.16;

library ArrayUtils {
  // internal functions can be used in internal library functions because
  // they will be part of the same code context
  function map(uint[] memory self, function (uint) pure returns (uint) f)
    internal
    pure
    returns (uint[] memory r)
  {
    r = new uint[](self.length);
    for (uint i = 0; i < self.length; i++) {
      r[i] = f(self[i]);
    }
  }
  function reduce(
    uint[] memory self,
    function (uint, uint) pure returns (uint) f
  )
    internal
    pure
    returns (uint r)
  {
    r = self[0];
    for (uint i = 1; i < self.length; i++) {
      r = f(r, self[i]);
    }
  }
  function range(uint length) internal pure returns (uint[] memory r) {
    r = new uint[](length);
    for (uint i = 0; i < r.length; i++) {
      r[i] = i;
    }
  }
}

contract Pyramid {
  using ArrayUtils for *;
```


```
  function pyramid(uint l) public pure returns (uint) {
    return ArrayUtils.range(l).map(square).reduce(sum);
  }
  function square(uint x) internal pure returns (uint) {
    return x * x;
  }
  function sum(uint x, uint y) internal pure returns (uint) {
    return x + y;
  }
}
```

下面的代码显示外部（external）函数类型的使用：

```
pragma solidity ^0.4.11;

contract Oracle {
  struct Request {
    bytes data;
    function(bytes memory) external callback;
  }
  Request[] requests;
  event NewRequest(uint);
  function query(bytes data, function(bytes memory) external callback) public
{
    requests.push(Request(data, callback));
    NewRequest(requests.length - 1);
  }
  function reply(uint requestID, bytes response) public {
    // Here goes the check that the reply comes from a trusted source
    requests[requestID].callback(response);
  }
}

contract OracleUser {
  Oracle constant oracle = Oracle(0x1234567); // known contract
  function buySomething() {
    oracle.query("USD", this.oracleResponse);
  }
  function oracleResponse(bytes response) public {
```

```
    require(msg.sender == address(oracle));
    // Use the data
  }
}
```

public 还是 external?

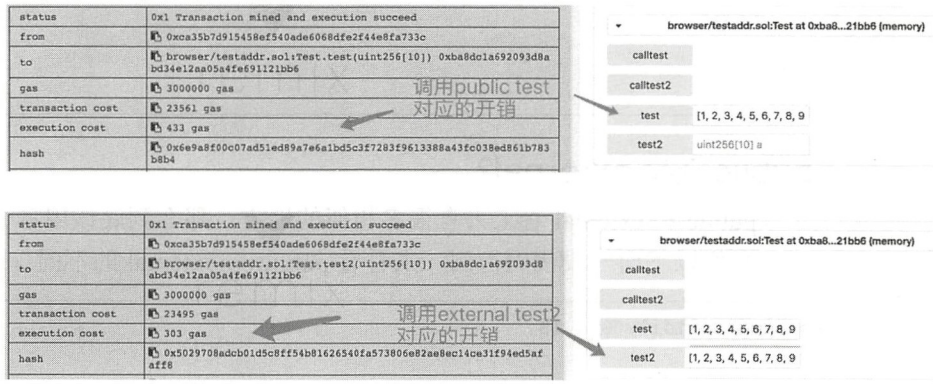
public 和 external 看上去有很多相似的地方，那么到底该用哪一个呢？

我们以下面一段代码为例，来看看 public 与 external 的不同：

```
pragma solidity^0.4.18;

contract Test {
    uint[10] x = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10];

    function test(uint[10] a) public returns (uint){
         return a[9]*2;
    }

    function test2(uint[10] a) external returns (uint){
         return a[9]*2;
    }

    function calltest() {
        test(x);
    }

    function calltest2() {
        this.test2(x);
        //test2(x);  //不能在内部调用一个外部函数，会报编译错误
    }

}
```

打开 Remix，贴入代码，创建合约。然后，分别调用 test 和 test2，对比执行花费的 gas。

可以看到调用 public 函数花销更大，这是为什么呢？



当使用 public 函数时，Solidity 会立即复制数组参数到内存，而 external 函数则是从 calldata 读取的，分配内存的开销比直接从 calldata 读取的开销要大得多。那为什么 public 函数要复制数组参数到内存呢？因为 public 函数可能会被内部调用，而内部调用数组的参数是当成指向一块内存的指针。对于 external 函数不允许内部调用，它直接从 calldata 读取数据，省去了复制的过程。

所以，如果确认一个函数仅仅在外部访问，那么请用 external 函数。

同样，我们接着对比 calltest()及 calltest2()，这里不截图了，大家自己运行对比一下就可以发现：calltest2 的开销比 calltest 的开销大得多，这是因为通过 this.f()模式调用，会有一个花费很大开销的 CALL 调用，并且它传参的方式也比内部传递的开销更大。

因此，当需要内部调用的时候，请用 public。

地址类型（Address）

在我们常见的编程语言里并没有地址这样的类型，但是地址类型在以太坊中却非常重要，因为以太坊的账户需要用地址来表示。

地址类型是一个值类型。

地址占用 20 字节，即以太坊地址的长度是 20 个字节。

可以用以下方式，声明一个地址类型：

```
address addr = 0xd25ed029c093e56bc8911a07c46545000cbf37c6;
```

地址类型也有成员。地址是所有合约的基础，其支持的运算符有：

- <=、<、==、!=、>=和>

> 目前，地址是所有合约的基础（基类），即合约也可以是一个类型并且继承自地址类型。不过官方文档说，从 Solidity 0.5.0 版本开始，合约将不再继承自地址类型，但会保留显式转换为地址。

地址类型的成员

- balance 属性及 transfer()函数

balance 用来查询账户余额，transfer()用来发送以太币（以 wei 为单位）。假设有 Address addr，那么 addr.balance 用来查询账户 addr 的余额，addr.transfer()用来向地址 addr 发送以太币（注意，很多人误以为 addr 是发送方，但实际上 addr 是接收方）。参数为以太币的数量，以 wei 为单位，即 1eth == 10 ** 18 wei。我们在后面第 5 章会详细介绍单位。

下面是一个简单的例子：

```
address x = 0x123;
address myAddress = this;
if (x.balance < 10 && myAddress.balance >= 10) x.transfer(10);
```

注解：如果 x 是合约地址，那么合约的回退函数（fallback）会随 transfer 调用一起执行（这个是 EVM 特性）。如果因 gas 耗光或其他原因失败，则转移交易会被还原，并且合约会出现异常并停止。

> 关于回退函数（fallback），简单来说就是合约中无函数名的函数。我们将在第 8 章介绍合约时做进一步阐释。

- send()函数

send 与 transfer 对应，但在底层上。如果执行失败，transfer 不会因异常停止，而 send 会返回 false。实际上 addr.transfer(y)与 require(addr.send(y))是等价的。

> 警告：send() 执行有一些风险。如果调用栈的深度超过 1024 或 gas 耗光，交易都会失败。

这在接收者是一个合约地址的情况下，尤其如此。因为 send()和 transfer()函数只会使用 2300 gas，这个 gas 通常只够发送一个事件的，大家可以在本节

结尾部分的代码中验证一下，在回退函数中多加入一些逻辑，然后向合约进行转账。

为了避免转账失败，一个方案是用 addr.call.value(y)()，这句代码在功能上等价于 addr.transfer(y)。使用这种方式会利用上之前所有可用的 gas，这可能会引发另一个可重入问题（我们会在第 11 章智能合约最佳实践中介绍）。

因此，为了保证安全，必须检查 send 的返回值。如果交易失败，则会退回以太币。

- call()、delegatecall()和 callcode()函数

为了和非 ABI 协议的合约进行交互，可以使用 call()函数。它可以向另一个合约发送原始数据，支持任何类型、任意数量的参数。每个参数会按规则（ABI 协议）打包成 32 字节并一一拼接到一起。只有一个例外是，如果第一个参数恰好为 4 个字节，那么会被认为是 ABI 协议定义的函数器指定的函数签名而直接使用。如果仅想发送消息体，则需要避免第一个参数是 4 个字节的情况。例如下面的例子：

```
address nameReg = 0x72ba7d8e73fe8eb666ea66babc8116a41bfb10e2;
nameReg.call("register", "MyName");
nameReg.call(bytes4(keccak256("fun(uint256)")), a);
```

call 函数返回一个 bool 值，以表明执行成功与否。正常结束返回 true，异常终止返回 false。但无法获取到结果数据，因为需要提前知道返回的数据的编码和数据大小（因为不知道对方使用的协议格式，所以不会知道返回的结果如何解析）。

提供.gas()修饰器进行调用：

```
namReg.call.gas(1000000)("register", "MyName");
```

还可以提供附带以太币：

```
nameReg.call.value(1 ether)("register", "MyName");
```

修饰器可以混合使用，修饰器调用顺序无所谓：

```
nameReg.call.gas(1000000).value(1 ether)("register", "MyName");
```

> 注解：目前还不能在重载函数上使用 gas 或 value 修饰符。

另外，我们还可以使用 delegatecall()。它与 call 方法的区别在于，仅仅是代码会被执行，而其他方面如存储、余额等都是用的当前合约的数据。使用 delegatecall()方法的目的是用来执行另一个合约中的库代码。所以开发者需要保证两个合约中的存储变量能兼容，以保证 delegatecall()能顺利执行。在 homestead 阶段之前，仅有一个受限的 callcode()方法可用，但 callcode 未提供对 msg.sender、msg.value 的访问权限。

上面的这三个方法 call()、delegatecall()、callcode()都是底层的消息传递调用，建议仅在万不得已时再使用，因为它们破坏了 Solidity 的类型安全。.gas()在 call()、callcode()和 delegatecall()函数下都可以使用，但 delegatecall()不支持.value()。

> 注解：所有合约都继承了 address 的成员，因此可以使用 this.balance 查询余额。不再鼓励使用 callcode，因为它以后可能会被移除。

> 警告：上述的函数都是底层的函数，使用时要异常小心。当调用一个未知的、可能是恶意的合约时，相当于把控制权交给了对方，它可能会调回你的合约。所以要准备好在调用返回时，应对合约的状态变量可能被恶意篡改的情况。

以下一段代码示例中演示了回退函数、 .balance 、transfer()、call()等用法，大家最好实际操作一下。

```
pragma solidity ^0.4.0;

contract AddrTest{
    event logdata(bytes data);
    function() payable {
        logdata(msg.data);
    }

    function getBalance() returns (uint) {
        return this.balance;
    }

    uint score = 0;
```

```
    function setScore(uint s) public {
        score = s;
    }

    function getScore() returns ( uint){
        return score;
    }
}

contract CallTest{
    function deposit() payable {
    }

    event logSendEvent(address to, uint value);
    function transferEther(address towho) payable {
        towho.transfer(10);
        logSendEvent(towho, 10);
    }

    function callNoFunc(address addr) returns (bool){
        return addr.call("tinyxiong", 1234);
    }

    function callfunc(address addr) returns (bool){
        bytes4 methodId = bytes4(keccak256("setScore(uint256)"));
        return addr.call(methodId, 100);
    }

    function getBalance() returns (uint) {
        return this.balance;
    }
}
```

如何区分合约地址及外部账号地址

在编程的时候，我们经常需要区分一个地址是合约地址还是外部账号地址，区分的关键是看这个地址有没有与之相关联的代码。

EVM 提供了一个操作码 EXTCODESIZE，用来获取地址相关联的代码大小

（长度），如果是外部账号地址，则没有代码返回。因此我们可以使用以下方法判断合约地址及外部账号地址：

```
function isContract(address addr) internal view returns (bool) {
  uint256 size;
  assembly { size := extcodesize(addr) }
  return size > 0;
}
```

如果是在合约外部判断，则可以使用 web3.eth.getCode()，或者是对应的 JSON-RPC 方法 eth_getcode。

getCode()用来获取参数地址所对应合约的代码，如果参数是一个外部账号地址，则返回"0x"；如果参数是合约，则返回对应的字节码，如下所示：

```
> web3.eth.getCode("0xa5Acc472597C1e1651270da9081Cc5a0b38258E3")
"0x"

web3.eth.getCode("0xd5677cf67b5aa051bb40496e68ad359eb97cfbf8")
>
"0x600160008035811a818181146012578301005b601b6001356025565b8060005260206000f
25b600060078202905091905056"
```

这样我们就可以通过 getCode()的内容判断是哪一种地址了。

地址常量（Address Literal）

一个能通过地址合法性检查（Address Checksum Test）的十六进制常量，就会被认为是地址，如 0xdCad3a6d3569DF655070DEd06cb7A1b2Ccd1D3AF。而不能通过地址合法性检查的 39 到 41 位长的十六进制常量，则会给出一个警告，将其视为普通的有理数常量。

> 地址合法性检查定义在 EIP-55 中，地址为 https://github.com/ethereum/EIPs/blob/master/EIPS/eip-55.md。

数据位置（Data Location）

前面我们已经介绍完了值类型，接下来介绍引用类型。但是在介绍引用类型之前，我们需要介绍一下数据位置，因为引用类型是一个复杂类型，占用的空间通常超过 256 位，拷贝时开销很大。因此我们需要考虑将它们存储在什么位置，是 memory（内存中，数据不是永久存在的）还是 storage（永久存储在区块链中）中。所有的复杂类型如数组和结构体都有一个额外属性：**数据的存储位置**，即 memory 或 storage。

根据上下文的不同，大多数情况下数据位置有默认值，可以通过指定关键字 memory 或 storage 修改它。

函数参数（包括返回的参数）默认存储位置是 memory。局部复杂类型变量（local variables）和状态变量（state variables）默认存储位置是 storage。

> 局部变量：局部作用域（越过作用域即不可被访问，等待被回收）的变量，如函数内的变量。状态变量：合约内声明的公有变量。

还有一个存储位置是 calldata，用来存储函数参数，是只读的、不会永久存储的数据位置。外部函数的参数（不包括返回参数）被强制指定为 calldata。效果与 memory 差不多。

数据的存储位置非常重要，因为他们影响着赋值行为。在 memory 和 storage 之间或与状态变量之间相互赋值，总是会创建出一个完全独立的拷贝。而将一个 storage 的状态变量，赋值给一个 storage 的局部变量，则是通过引用传递完成的。所以对于局部变量的修改，要同时修改关联的状态变量。另一方面，将一个 memory 的引用类型赋值给另一个 memory 的引用，是不会创建拷贝的，即 memory 之间是通过引用传递完成的。

> 注意：
> 1. 不能将 memory 赋值给局部变量。
> 2. 对于值类型，总是会进行拷贝的。

下面看一段代码：

```
pragma solidity ^0.4.0;

contract C {
    uint[] x; //x 的存储位置是 storage

    // memoryArray 的存储位置是 memory
    function f(uint[] memoryArray) public {
        x = memoryArray;    // 从 memory 复制到 storage
        var y = x;          // storage 引用传递局部变量 y（y 是一个 storage 引用）
        y[7];               // 返回第 8 个元素
        y.length = 2;       // x 同样会被修改
        delete x;           // y 同样会被修改

        // 错误，不能将 memory 赋值给局部变量
        // y = memoryArray;

        // 错误，不能通过引用销毁 storage
        // delete y;

        g(x);               // 引用传递，g 可以改变 x 的内容
        h(x);               // 拷贝到 memory，h 无法改变 x 的内容
    }

    function g(uint[] storage storageArray) internal {}
    function h(uint[] memoryArray) public {}
}
```

强制指定的数据位置（Forced data location）

- 外部函数的参数（不包括返回参数）强制指定的数据位置为 calldata。
- 状态变量强制指定的数据位置为 storage。

默认的数据位置（Default data location）

- 函数参数及返回参数默认的数据位置为 memory。
- 复杂类型的局部变量默认的数据位置为 storage。

storage 存储结构是在合约创建的时候就确定好的，它取决于合约所声明的状态变量。但是内容可以被调用（交易）改变。

> Solidity 称这个为状态改变，这也是合约级变量称为**状态变量**的原因。也可以简单理解为状态变量都是 storage 存储。

memory 只能用于函数内部，memory 声明用来告知 EVM 在运行时创建一块（固定大小）内存区域给变量使用。

> storage 在区块链中是用 key/value 的形式存储的，而 memory 则表现为字节数组。

关于栈（stack）

EVM 是一个基于栈的语言，而栈实际上是内存（memory）中的一个数据结构。每个栈元素为 256 位，栈的最大长度为 1024。

值类型的局部变量是存储在栈上的。

不同存储的 gas 消耗

- storage 会永久保存合约状态变量，开销最大。
- memory 仅保存临时变量，函数调用之后释放，开销很小。
- stack 保存很小的局部变量，几乎免费使用，但有数量限制。

数组（Array）

数组可以在声明时指定长度，也可以动态变长。一个元素类型为 T，固定长度为 k 的数组，可以声明为 T[k]；而一个动态变长的数组，可以声明为 T[]。

```
uint [10] tens;

uint [] us;
```

带初始化的声明方式如下：

```
uint [] public u = [1, 2, 3];

string[4] adaArr = ["This", "is", "an", "array"];
```

或者用 new 关键字进行声明，形式如下：

```
uint[] c = new uint[](7);

bytes public _data = new bytes(10);

string [] adaArr1 = new  string[](4);
```

数组通过下标进行访问，序号是从 0 开始的。例如访问第 1 个元素时使用 tens[0]，对元素赋值 tens[0] = 1。

还可以声明一个多维数组。例如声明一个类型为 uint、长度为 5 的变长数组（5 个元素都是变长数组），则可以声明为 uint[][5]。

> 注意，在其他语言中，多维数组的长度声明是反的。比如用 Java 声明一个包含 5 个元素、每个元素都是数组的方式为 int[5][]。

要访问第 3 个动态数组的第 2 个元素，使用 x[2][1]即可。数组的序号是从 0 开始的，序号顺序与定义相反。数组访问的方式和非区块链语言一致。

对存储在 storage 的数组来说，元素类型可以是任意的，类型可以是数组、映射类型、结构体等。但对于存储在 memory 的数组来说，如果它是 public 函数的参数，则不能是映射类型的数组，只能是支持 ABI 的数组类型（参考第 10 章）。

类型为数组的 public 状态变量，Solidity 编译器会创建一个访问器，如果要访问数组的某个元素，指定数字下标就好了。下面是一段示例代码：

```
pragma solidity ^0.4.0;

contract C {

    uint [] public u = [1, 2, 3];    // 生成访问器
    string  s = "abcdefg";

    uint[] c;  //storage
    function g(){
        c = new uint[](7);
        c.length = 10;   //可以修改 storage 的数组
        c[9] = 100;
    }
```

```
    function h() public  returns (uint) {
        return bytes(s).length;
    }

    function f() public  returns (byte) {
        return bytes(s)[1];     // 转为数组访问
    }

}
```

创建内存数组

可以使用 new 关键字创建一个存储在 memory 上的数组。与存储在 storage 上的数组不同的是，该数组不能通过成员.length 的值来修改数组的大小属性。我们来看看下面的例子：

```
pragma solidity ^0.4.16;

contract C {
    function f(uint len) public pure {
        uint[] memory a = new uint[](7);

        //a.length = 100;  // 错误
        bytes memory b = new bytes(len);
        // 这里 a.length == 7 and b.length == len
        a[6] = 8;
    }

}
```

数组常量及内联数组

数组常量，是一个数组表达式（还没有赋值到变量）。下面是一个简单的例子：

```
pragma solidity ^0.4.16;

contract C {
    function f() public pure {
        g([uint(1), 2, 3]);
```

```
    }
    function g(uint[3] _data) public pure {
        // ...
    }
}
```

通过数组常量，创建的数组是存储在 memory 的，同时其长度也是固定的。元素类型则是使用刚好能存储的元素，比如[1,2,3]只需要为 uint8 即可存储，它的类型是 uint8[3] memory。

由于 g()方法的参数需要的是 uint（默认的 uint 表示的其实是 uint256），所以需要对第一个元素进行类型转换，使用 uint(1)来进行这个转换。

还需注意的一点是，定长数组不能与变长数组相互赋值，我们来看下面的错误代码示例：

```
// 无法编译
pragma solidity ^0.4.0;

contract C {
    function f() public {
        // The next line creates a type error because uint[3] memory
        // cannot be converted to uint[] memory
        uint[] x = [uint(1), 3, 4];
    }
}
```

不过，Solidity 已经计划在未来移除这样的限制。当前是因为 ABI 传递数组，所以还有些问题。

数组成员

length 属性

数组有一个.length 的成员属性，表示当前的数组长度。对于存储在 storage 的变长数组，可以通过给.length 赋值调整数组长度。而存储在 memory 的变长数组不支持修改.length 调整数组长度。

注意，不能通过访问超出当前数组长度的方式，来自动实现改变数组长度。存储在 memory 的数组虽然可以通过参数灵活指定长度，但一旦创建，长度便

不可调整。

push 方法

存储在 storage 的变长数组和 bytes 都有一个 **push 成员**方法（string 没有），用于附加新元素到数据末端，返回值为新的长度。

注意，string 没有 push 方法，存储在 memory 的数组也不支持 push。

数组的限制

当前在外部函数中，不能使用多维数组。

另外，基于 EVM 的限制，不能通过外部函数返回动态的内容。

```
contract C {
     function f() returns (uint[]) { ... }
      }
```

在这个例子中，通过 Web3.js 调用能返回数据，但从 Solidity 中调用则不能返回数据。一种绕过这个问题的方法是使用一个非常大的静态数组。

```
pragma solidity ^0.4.16;

contract ArrayContract {
    uint[2**20] m_aLotOfIntegers;
    // 这里不是两个动态数组的数组，而是在一个动态数组里，每个元素是长度为 2 的数组
    bool[2][] m_pairsOfFlags;
    // newPairs 存在 memory 里，因为是函数参数
    function setAllFlagPairs(bool[2][] newPairs) public {
        m_pairsOfFlags = newPairs;
    }

    function setFlagPair(uint index, bool flagA, bool flagB) public {
        // 访问不存在的 index 会抛出异常
        m_pairsOfFlags[index][0] = flagA;
        m_pairsOfFlags[index][1] = flagB;
    }

    function changeFlagArraySize(uint newSize) public {
        // 如果新 size 更小，移除的元素会被销毁
        m_pairsOfFlags.length = newSize;
```

```
    }

    function clear() public {
        // 销毁
        delete m_pairsOfFlags;
        delete m_aLotOfIntegers;
        // 同销毁一样的效果
        m_pairsOfFlags.length = 0;
    }

    bytes m_byteData;

    function byteArrays(bytes data) public {
        // byte arrays ("bytes") are different as they are stored without padding,
        // but can be treated identical to "uint8[]"
        m_byteData = data;
        m_byteData.length += 7;
        m_byteData[3] = byte(8);
        delete m_byteData[2];
    }

    function addFlag(bool[2] flag) public returns (uint) {
        return m_pairsOfFlags.push(flag);
    }

    function createMemoryArray(uint size) public pure returns (bytes) {
        // Dynamic memory arrays are created using "new":
        uint[2][] memory arrayOfPairs = new uint[2][](size);
        // Create a dynamic byte array
        bytes memory b = new bytes(200);
        for (uint i = 0; i < b.length; i++)
            b[i] = byte(i);
        return b;
    }
}
```

字符串 string 及字节数组 bytes

bytes 和 string 是一种特殊的数组。它们的声明几乎是一样的，形式如下：

```
bytes bs;
    bytes bs0 = "12abcd";
    bytes bs1 = "abc\x22\x22";   // 十六进制数
    bytes bs2 = "Tiny\u718A";   // 718A 为汉字“熊”Unicode 编码值

    string str0;
    string str1 = "TinyXiong\u718A";
```

bytes 是动态分配大小字节数组，bytes 类似于 byte[]，但在外部函数作为参数调用中，bytes 会进行压缩打包。string 类似于 bytes，但目前不提供长度和按序号的访问方式。所以应该尽量使用 bytes 而不是 byte[]。

bytes 和 string 都可以用来表达字符串，他们的区别是 bytes 用来存储任意长度的字节数据，string 用来存储任意长度（UTF-8 编码）的字符串数据。

可以将字符串 s 通过 bytes(s)转为一个 bytes，通过 bytes(s).length 获取长度，bytes(s)[n]获取对应的 UTF-8 编码。通过下标访问获取到的不是对应字符，而是 UTF-8 编码，比如中文的编码是变长的多字节，因此通过下标访问中文字符串得到的只是其中的一个编码。

string 扩展

Solidity 语言本身提供的 string 功能比较弱，因此有人实现了 string 的实用工具库 stringutils，GitHub 地址为 https://github.com/Arachnid/solidity-stringutils，并且在这个库中引入了一个 slice 的概念，下面列举了几个使用示例。

- 获取字符串长度

```
var len = "Unicode snowman ☃".toSlice().len();  // 17
```

- 获取第一个单词

```
var s = "foo bar baz".toSlice();

var firstWord = s.split(" ".toSlice());
```

- 查找第一个匹配字符的字符串

```
var s = "A B C B D".toSlice();

s.rfind("B".toSlice()); // "A B C B"
```

- 把字符串转成字符串数组

```
function smt() {
    var s = ""This-Is-A-Problem"".toSlice();
    var delim = "-".toSlice();
    var parts = new string[](s.count(delim) + 1);

    for(uint i = 0; i < parts.length; i++) {
       parts[i] = s.split(delim).toString();
    }
   }
```

另外，如判断以什么字符开头的 startsWith()函数、判断以什么字符结尾的 endsWith()函数及实现字符串拼接的 concat()函数等都在这个工具库中。

不过当前的 stringutils 并没有大小写转换工具，这里给读者提供一个方法：

```
function _toLower(string str) internal returns (string) {
       bytes memory bStr = bytes(str);
       bytes memory bLower = new bytes(bStr.length);
       for (uint i = 0; i < bStr.length; i++) {
            // 大写字母
            if ((bStr[i] >= 0x41) && (bStr[i] <= 0x5A)) {
                 // 加 0x20 变为小写
                 bLower[i] = bytes1(int(bStr[i]) + 32);
            } else {
                 bLower[i] = bStr[i];
            }
       }
       return string(bLower);
  }
```

定长字节数组还是字符串

有时定长字节数组会用来代替字符串使用，这是为什么呢？我们先来对比以下不同函数的 gas 消耗：

```
pragma solidity ^0.4.18;
contract compGas {
    string constant ss = "Tiny Xiong";
    bytes32 constant bt32 = "Tiny Xiong";
    function  getString() payable public  returns(string) {
        return ss;
    }

    function  getByte32() payable public returns(bytes32) {
        return bt32;
    }
}
```

我在本地测试时，getByte32()消耗了 21490 gas，getString()消耗了 21853 gas。相比变长的 string，定长字节数组 gas 消耗更少。因此，如果字符串的长度是固定的（或长度可以确定），尽量使用 bytes32（官方文档里介绍的是尽量使用定长的如 bytes1 到 bytes32 中的一个，因为更省空间，我的个人经验是使用 bytes32 消耗的 gas 最少）。

结构体（Struct）

Solidity 使用 struct 关键字来定义自定义类型，自定义的类型是引用类型。struct 除可以使用基本类型作为成员以外，还可以使用数组、结构体、映射作为成员。

结构体类型定义语法如下：

```
struct CustomType {
    bool myBool;
    uint myInt;
}

// 使用复杂类型作为自定义结构体元素
struct CustomType2 {
    CustomType[] cts;
    mapping(string=>CustomType) indexs;
}
```

不能在声明一个 struct 的同时将自身 struct 作为成员，因为结构体的大小必须是有限的。

但**struct**可以作为**mapping**的值类型成员，如：

```
// 非法
struct CustomType {
    bool myBool;
    uint myInt;
    CustomType aa;
}

// 有效
struct CustomType2 {
    CustomType[] cts;
    mapping(string=>CustomType2) indexs;
}

struct CustomType3 {
    string name;
    mapping(string=>uint) score;
    int age;
}
```

声明与初始化

使用结构体声明变量及初始化有以下几个方式。

1. 仅声明变量而不初始化，此时会使用默认值创建结构体变量，例如：

```
CustomType ct1;
```

2. 按成员顺序（结构体声明时的顺序）初始化，例如：

```
CustomType ct1 = CustomType(true, 2);          // 只能作为状态变量这样使用
CustomType memory ct2 = CustomType(true, 2);   // 在函数内声明
```

这种方式需要特别注意参数的类型及数量的匹配。另外，如果结构体中有 mapping，则需要跳过对 mapping 的初始化。例如对上面 CustomType3 的初始化方法为：

```
CustomType3 memory ct = CustomType3("tiny", 2);
```

3．命名方式初始化。

使用命名方式可以不按定义的顺序初始化，初始化方法如下：

```
// 使用命名变量初始化
CustomType memory ct = CustomType({ myBool: true, myInt: 2});
```

参数的个数需要保持和定义时一致，如果有 mapping 类型，则同样需要忽略。

我们看看下面有关 struct 使用得较为完整的例子：

```
pragma solidity ^0.4.11;
contract CrowdFunding {
    // 定义一个包含两个成员的新类型
    struct Funder {
        address addr;
        uint amount;
    }

    struct Campaign {
        address beneficiary;
        uint fundingGoal;
        uint numFunders;
        uint amount;
        mapping (uint => Funder) funders;
    }

    uint numCampaigns;
    mapping (uint => Campaign) campaigns;

    function newCampaign(address beneficiary, uint goal) public returns (uint
campaignID) {
        campaignID = numCampaigns++; // campaignID 作为一个变量返回
        // 创建一个结构体实例，存储在 storage 上，放入 mapping 里
        campaigns[campaignID] = Campaign(beneficiary, goal, 0, 0);
    }
```

```
    function contribute(uint campaignID) public payable {
        Campaign storage c = campaigns[campaignID];
        // 用 mapping 对应项创建一个结构体引用
        // 也可以用 Funder(msg.sender, msg.value) 来初始化
        c.funders[c.numFunders++] = Funder({addr: msg.sender, amount:
msg.value});
        c.amount += msg.value;
    }

    function checkGoalReached(uint campaignID) public returns (bool reached) {
        Campaign storage c = campaigns[campaignID];
        if (c.amount < c.fundingGoal)
            return false;
        uint amount = c.amount;
        c.amount = 0;
        c.beneficiary.transfer(amount);
        return true;
    }
}
```

上面是一个简化版的众筹合约，但它可以让我们理解 struct 的基础概念。struct 可以用于映射和在数组中作为元素，其本身也可以包含映射和数组等类型。

注意在函数中，将一个 struct 赋值给一个局部变量（默认是 storage 类型），实际是拷贝的引用，所以修改局部变量值的同时会影响到原变量。

结构体也可以直接通过访问成员来修改成员的值（不用赋值到一个局部变量来进行修改），比如：campaigns[campaignID].amount = 0。

结构体限制

结构体目前仅支持在合约内部使用或继承合约内使用，如果要在参数和返回值中使用结构体，函数必须声明 internal，例如：

```
    // 合法
    function interFunc(CustomType ct) internal {
    }

    // 非法
    function exterFunc(CustomType ct) public {
```

```
}
```

不过上面的 exterFunc 函数已经在 ABIEncoderV2 试验版本中开始支持了。启用 ABIEncoderV2 的方法是在源文件开头加入：pragma experimental ABIEncoderV2。

目前，如果想在合约之间传递结构体，则必须对结构体成员进行拆解才行。

映射（Mapping）

映射类型和 Java 的 Map 或 Python 的 Dict 功能差不多，它是一种键值对的映射关系存储结构，定义方式为 mapping(_KeyType => _KeyValue)。如：

```
mapping( uint => string) idName;
```

映射是一种使用广泛的类型，经常在合约中充当一个类数据库的角色，比如在代币合约中用映射来存储账户的余额，在游戏合约里可以用映射来存储每个账号的级别，如：

```
mapping(address => uint) public balances;
mapping(address => uint) public userLevel;
```

映射的访问和数组类似，可以用 balances[userAddr]访问。

键类型允许除映射、变长数组、合约、枚举、结构体之外的几乎所有类型访问。值类型则没有任何限制，可以为任何类型包括映射类型。

映射可以作为一个哈希表，所有可能的键都会被虚拟化创建，映射到一个类型的默认值（二进制数全零表示）。在映射表中，并不存储键的数据，仅存储它的 keccak256 哈希值，这个哈希值在查找值时需要用到。正因为如此，映射是没有长度的，也没有键集合或值集合的概念。

可以通过将映射标记为 public，来让 Solidity 创建一个访问器。通过提供一个键值作为参数来访问它并返回对应的值。映射的值类型也可以是映射，使用访问器访问时，要提供这个映射值所对应的键，不断重复这个过程。来看一个例子：

```
pragma solidity ^0.4.0;

contract MappingExample {
    mapping(address => uint) public balances;

    function update(uint newBalance) public {
        balances[msg.sender] = newBalance;
    }
}

contract MappingUser {
    function f() public returns (uint) {
        MappingExample m = new MappingExample();
        m.update(100);
        return m.balances(this);
    }
}
```

限制

映射类型仅能用来作为状态变量，或者在内部函数中作为 storage 类型的引用。

映射并未提供迭代输出的方法，即不能通过遍历访问所有元素，也无法获得所有键或值的列表。如果需要使用，我们可以自行实现一个这样的数据结构。参考以下链接，https://github.com/ethereum/dapp-bin/blob/master/library/iterable_mapping.sol。

类型转换

和其他大部分语言一样，Solidity 也支持类型转换，例如将一个数字字符串转换为整型或浮点数。转换可分为隐式转换和显式转换。

隐式转换

如果运算符支持两种不同的类型，那么编译器会尝试隐式转换类型，同理，赋值时也是类似的。通常，隐式转换需要保证不丢失数据且语义可通。如 uint8 可以转化为 uint256（位数更多，数据不会丢失），但 int8 不能转为 uint256，因

为 uint256 不能表示–1。

此外，任何无符号整数都可以转换为相同或更大长度的字节数组。再者，任何可以转换为 uint16 的类型，也可以转换为 address 类型。

下面是一个示例：

```
function add() public pure returns (uint) {
        uint8 i = 10;
        uint16 j = 20;

        uint16 k = i + j;

        return k;
    }
```

在进行 uint16 k = i + j 运算时，i 会隐式转换为 uint16。在 return k 时，k 会隐式转换为 uint256。

显式转换

如果编译器不允许隐式的自动转换，但你知道转换没有问题时，可以进行强转。需要注意的是，不正确的转换会带来错误，所以要进行谨慎测试。例如下面的例子中把一个负的 int8 转换为 uint。

```
pragma solidity ^0.4.0;

contract caseTest{
    uint a;

    function f() returns (uint){
      int8 y = -3;
      uint x = uint(y);
      return x;   // 0xfffff..fd  (64 个十六进制数)
    }
}
```

如果转换为一个更小的类型，那么高位字符将被截断。

```
uint32 a = 0x12345678;
uint16 b = uint16(a); // b = 0x5678
```

var 类型推导

尽管 var 在很多文档中还有介绍，但是在 Solidity 0.4.20 及之后的编译器版本中，var 关键字已经被标记为弃用（deprecated），因此本节的内容读者仅了解一下即可。

有时为了方便，不需要明确指定一个变量的类型。编译器会通过第一个向这个对象赋予的值的类型来进行推导以确定类型，如下所示：

```
uint24 x = 0x123;
var y = x;
```

此时 y 的类型是 uint24。

函数的参数包括返回参数，不可以使用 var 这种不指定类型的方式。

```
function f() public pure returns (uint, bool, uint) {
       return (7, true, 2);
}
var (x,y,z) = f(); //返回多个值

(x,y) = (y,x)  // 交换值
```

有一个地方需要注意，由于类型推导是根据第一个变量进行的赋值，所以代码 for (var i = 0; i < 2000; i++) {}将是无限循环的，因为一个 uint8 的 i 将小于 2000。

运算符

运算符优先级如下表所示。

表 1　运算符优先级

优先级	描述	运算符
1	后缀加、减	++　--
1	New 表达式	new
1	数组下标访问	<array>[<index>]
1	成员下标访问	<object>.<member>
1	函数访问	| <func>(<args...>)

续表

优先级	描述	运算符
1	括号	(<statement>)
2	后缀自加、自减	++ —
2	一元加、一元减	+ –
2	delete 运算符	delete
2	逻辑反	!
2	按位反	~
3	幂	**
4	乘、除、余	* / %
5	加、减	+ –
6	位移	<< >>
7	按位与	&
8	按位异或	^
9	按位或	|
10	不等于运算符	< > <= >=
11	等于运算符	== !=
12	逻辑与	&&
13	逻辑或	||
14	条件表达式	<conditional> ? <if-true> : <if-false>
15	赋值运算符	= |= ^= &= <<= >>= += –= *= /= %=
16	逗号操作符	,

delete 操作符

用过 C 或 C++的人应该使用过 delete 来释放变量的空间。在 Solidity 中，delete 操作符的功能尽管也可以释放空间，但 delete 操作符更像是将某个变量重置为初始值，例如 delete a 对于整数 a，效果等同于 a = 0。

如果 delete 作用到数组上，则是把数组中的每个元素设置为初始值。变长数组则是将长度设置为 0。对于结构体也是一样的，是将所有的成员均重置为初始值。代码示例如下：

```
bool b = true;
    delete b;  // b = false;

    uint a = 1;
    delete a ; // a =  0

    address addr = msg.sender;
    delete addr ; // addr = 0x0

    bytes  memory bs = "123";
    delete bs; // bs = 0x0

    string memory str = "tiny xiong";
    delete  str; // str = ""

    // 变长数组
    uint[] memory arr = new uint[](7);
    delete arr;  // a.length = 0;

    struct CustomType {
        bool myBool;
        uint myInt;
    }

    CustomType memory ct = CustomType(true, 100);
    delete ct;  // ct.myBool = false ; myInt = 0
```

delete 对于映射类型几乎无影响，因为键可能是任意的且往往不可知。所以，如果你删除一个结构体，它会递归删除所有非 mapping 的成员。当然，你可以单独删除映射里的某个键，以及这个键映射的某个值。例如：

```
mapping(address=>uint) balances;

    function deleteMap() public {
    // 非法
        delete balances;

    // 正确
```

```
        delete balances[msg.sender];
    }
```

如果删除一个结构体，那么它会递归删除所有非 mapping 的成员。

我们来看看下面的示例：

```
pragma solidity ^0.4.0;

contract DeleteExample {
    uint data;
    uint[] dataArray;

    function f() public {
      //值传递
        uint x = data;
        delete x;     //删除 x 不会影响 data
        delete data; // 删除 data 同样不会影响 x，因为是值传递，保存的是一份原值拷贝

          //引用赋值
        uint[] storage y = dataArray;
        //删除 dataArray 会影响 y，y 将被赋值为初值
        delete dataArray;


        //报错，因为删除的是一个赋值操作，所以不能向引用类型的 storage 直接赋值
        //delete y;

    }
}
```

通过上面的代码，我们可以看出对于值类型是值传递的，删除 x 不会影响到 data。同样，删除 data 也不会影响到 x。因为它们都存了一份原值的拷贝。

而对于复杂类型略有不同，复杂类型在赋值时使用的是引用传递，删除会影响所有相关变量。比如在上述代码中，删除 dataArray 同样会影响到 y。

由于 delete 的行为更像是赋值操作，所以不能在上述代码中执行 delete y。我们不能对一个 storage 的引用赋值。

另外，关于 delete 的 gas 消耗有一个看起来矛盾的地方，即在清理空间的时候是可以获得 gas 的返还的。但因为 delete 操作本身消耗 gas，所以在实际使

用时最好进行 gas 消耗的对比。

本章小结

类型是本书的重点之一，在编写程序时，离不开定义变量。作为静态类型的语言 Solidity，必须在定义变量时确定类型。本章对 Solidity 中的所有类型都做了介绍，同时介绍了类型转换、类型推导，以及运算符的优先级，方便大家在编程的时候查阅。

第 5 章

Solidity 中的单位

Solidity 中有两种单位：货币单位和时间单位，这一章详细介绍一下。

货币单位（Ether Unit）

一个数字常量（字面量）后面跟随一个后缀 wei、finney、szabo 或 ether，这个后缀就是货币单位。不同的单位可以转换，不含任何后缀的默认单位是 wei。不同的以太币单位转换关系如下：

- 1 ether == 10^3 finney == 1000 finney
- 1 ether == 10^6 szabo
- 1 ether == 10^{18} wei

> 插曲：以太币单位其实是密码学家们的名字，以太坊创始人为了纪念他们在数字货币领域的贡献而设。他们分别是：
> wei，Wei Dai（戴伟），密码学家，发表了 B-money。
> finney，Hal Finney（芬尼），密码学家，提出了工作量证明机制（PoW）。
> szabo，Nick Szabo（尼克萨博），密码学家，智能合约的提出者。

我们可以使用以下代码验证一个转换关系：

```
pragma solidity ^0.4.16;

contract testUnit {
    function tf() public pure returns (bool) {
      if (1 ether == 1000 finney){
          return true;
      }
      return false;
    }

    function ts() public pure returns (bool) {
      if (1 ether == 1000000 szabo){
          return true;
      }
      return false;
    }

    function tgw() public pure returns (bool) {
      if (1 ether == 1000000000000000000 wei){
          return true;
      }
      return false;
    }
}
```

注意在 Solidity 智能合约里只有 4 个单位：wei、finney、szabo 和 ether。不过在一些其他场合还有一些衍生的单位如：Kwei、Mwei、Gwei、Mether、Gether、Tether。这里给大家推荐一个货币转换小工具，其网址为 https://etherconverter.online/。

时间单位（Time Unit）

时间单位 seconds、minutes、hours、days、weeks、years 均可作为后缀，并进行相互转换，规则如下：

- 1 == 1 seconds（默认以 seconds 为单位）
- 1 minutes == 60 seconds

- 1 hours == 60 minutes
- 1 days == 24 hours
- 1 weeks = 7 days
- 1 years = 365 days

使用这些单位进行日期计算时需要特别小心，因为不是每年都有 365 天，并且不是每天都有 24 小时，例如还有闰秒。由于无法预测闰秒，所以必须由外部的预言（oracle）来更新从而得到一个精确的日历库。

这些后缀不能用于变量。如果想对输入的变量设置不同的单位，可以使用以下方式：

```
pragma solidity ^0.4.16;

contract testTUnit {

    function currTimeInSeconds() public pure returns (uint256){
        return now;
    }

    function f(uint start, uint daysAfter) public {
        if (now >= start + daysAfter * 1 days) {
        // ...
        }
    }
}
```

如果在合约中需要使用年、月、日等时间单位，则可以引入 DateTime 库。比如可以通过时间戳解析出对应的年、月、日信息，DateTime 库的 GitHub 地址为 https://github.com/pipermerriam/ethereum-datetime。

本章小结

本章介绍了 Solidity 中的货币单位、时间单位，以及各个单位之间是如何进行转换的。

第 6 章

Solidity 全局变量及函数

Solidity 全局变量及函数可以认为是 Solidity 提供的 API，这些全局变量及函数主要分为以下几类：

1. 有关区块和交易的属性
2. ABI 编码函数
3. 有关错误处理
4. 有关数学及加密功能
5. 地址相关
6. 合约相关

区块和交易的属性

Solidity 中有一些全局变量用来提供区块（或链）当前的信息，如下。

- blockhash（uint blockNumber）returns (bytes32)：返回给定区块号的哈希值，只支持最近的 256 个区块且不包含当前区块。在 Solidity 0.4.22 之前这个属性是 block. Blockhash(uint blockNumber)。
- block.coinbase (address)：当前块矿工的地址，括号中表示返回值的类型。
- block.difficulty (uint)：当前块的难度。
- block.gaslimit (uint)：当前块的 gaslimit。
- block.number (uint)：当前区块的块号。

- block.timestamp (uint)：当前块的 Unix 时间戳（从 1970/1/1 00:00:00 UTC 开始所经过的秒数）。
- gasleft() (uint256)：获取剩余 gas。
- msg.data (bytes)：完整地调用数据（calldata）。
- msg.gas (uint)：当前还剩的 gas。
- msg.sender (address)：当前调用发起人的地址。
- msg.sig (bytes4)：调用数据（calldata）的前四个字节（例如，函数标识符）。
- msg.value (uint)：这个消息所附带的以太币，单位为 wei。
- now (uint)：当前块的时间戳（block.timestamp 的别名）。
- tx.gasprice (uint)：交易的 gas 价格。
- tx.origin (address)：交易的发送者（全调用链）。

上面的属性在编写智能合约时经常会用到。对这些变量的使用也非常简单，其实在讲解第 1 章的货币示例代码时就已经使用过了，我们回顾一下代码：

```
constructor() public {
  minter = msg.sender;
}
function mint(address receiver, uint amount) public {
    if (msg.sender != minter) return;
    balances[receiver] += amount;
}
```

在构造函数和 mint()函数中，使用了 msg.sender 全局属性来获取调用发起人的地址。在构造函数中，使用 minter 变量保存了合约创建者（也是调用发起者）的地址，在 mint()函数中对 minter 和 msg.sender 进行对比，确保只有合约的创建者才可以成功调用 mint()函数（因为挖矿这个动作只有创建者才能执行，就像人民币只有央行才可以发行一样）。

其他属性的使用和 msg.sender 的用法是一样的，不再做过多介绍。

注意：

msg 的所有成员值，如 msg.sender、msg.value 的值可以因为每一次外部函数调用或库函数调用发生变化（因为 msg 就是和调用相关的全局变量）。

不应该依据 block.timestamp、now 和 block.blockhash 产生一个随机数（除非你确实需要这样做），因为这两个值在一定程度上被矿工影响（比如在赌博合

约里，不诚实的矿工可能会重新选择一个对自己有利的 hash）。

对于同一个链上连续的区块来说，当前区块的时间戳会大于上一个区块的时间戳。

为了可扩展性的原因，只能查最近的 256 个块，其他的将返回 0。

ABI 编码函数

Solidity 提供了以下函数，用来直接得到 ABI 编码信息。

- abi.encode(...) returns (bytes)：计算参数的 ABI 编码。
- abi.encodePacked(...) returns (bytes)：计算参数的紧密打包编码。
- abi. encodeWithSelector(bytes4 selector, ...) returns (bytes)：计算函数选择器和参数的 ABI 编码。
- abi.encodeWithSignature(string signature, ...) returns (bytes)：等价于 abi.encodeWithSelector(bytes4(keccak256(signature), ...)。

通过以上方法，函数可以在不被调用的情况下，获得 ABI 编码值。下面我们通过一段代码来看看这些方式的使用：

```
pragma solidity ^0.4.24;

contract testABI {
       function abiEncode() public constant returns (bytes) {
       abi.encode("string");  // 计算 string 的 ABI 编码
       return abi.encode("baz(uint,address)"); //计算函数 ABI 编码
    }
}
```

错误处理函数

Solidity 提供了全局函数用于错误处理，如下所示。

- assert（bool condition）：用于判断内部错误，条件不满足时抛出异常。
- require（bool condition）：用于判断输入或外部组件错误，条件不满足时抛出异常。
- require(bool condition, string message)：同上，只是多提供了一个错误信息。
- revert()：终止执行并还原改变的状态。
- revert(string reason)：同上，只是多提供了一个错误信息。

上面的 mint()方法，可以使用错误处理的方式进行改写，代码如下：

```
function mint(address receiver, uint amount) public {
        require (msg.sender == minter); //使用 require 进行条件判断
        balances[receiver] += amount;
    }
```

错误处理属于程序的控制结构。更多细节会在第 7 章控制结构章节中进一步介绍。

数学及加密功能

Solidity 提供的有数学与加密功能的函数有：

- addmod(uint x, uint y, uint k) returns (uint)：计算(x + y) % k，加法支持任意的精度且不会在2**256处溢出，从Solidity 0.5.0版本开始断言k != 0。
- mulmod(uint x, uint y, uint k) returns (uint)：计算 (x * y) % k，乘法支持任意的精度且不会在 2**256 处溢出，从 Solidity 0.5.0 版本开始断言 k != 0。
- keccak256(...) returns (bytes32)：使用 Keccak-256 计算 hash 值，为紧密打包参数。
- sha256(...) returns (bytes32)：使用 SHA-256 计算 hash 值，为紧密打包参数。
- sha3(...) returns (bytes32)：keccak256 的别名。
- ripemd160(...) returns (bytes20)：使用 RIPEMD-160 计算 hash 值，为紧密打包参数。
- ecrecover(bytes32 hash, uint8 v, bytes32 r, bytes32 s) returns (address)：通过椭圆曲线签名来恢复与公钥关联的地址，或者在错误时返回零。可用于签名数据的校验，如果返回结果是签名者的公匙地址，那么说明数据是正确的。

> ecrecover 函数需要四个参数，需要被签名数据的哈希结果值，r、s、v 分别来自签名结果串。
> r= signature[0:64]
> s= signature[64:128]
> v= signature[128:130]
> 其中 v 取出来的值或者是 00 或者是 01。要使用时，我们先要将其转为整型，再加上 27，所以我们将得到 27 或 28。在调用函数时 v 将填入 27 或 28。

用 JavaScript 表达如下：

```
var msg = '0x8CbaC5e4d803bE2A3A5cd3DbE7174504c6DD0c1C'

var hash = web3.sha3(msg)
var sig = web3.eth.sign(address, h).slice(2)
var r = `0x${sig.slice(0, 64)}`
var s = `0x${sig.slice(64, 128)}`
var v = web3.toDecimal(sig.slice(128, 130)) + 27
```

紧密打包参数（tightly packed）意思是说参数不会补位，而是直接连接在一起的，例如：

```
keccak256("ab", "c")
keccak256("abc")

keccak256(0x616263)  // hex
keccak256(6382179)
keccak256(97, 98, 99)   //ascii
```

如果需要填充，可以使用显式类型转换：keccak256("\x00\x12") 与 keccak256(uint16(0x12))相同。

注意，常量将使用存储它们所需的最少字节数来打包，例如 keccak256(0) == keccak256(uint8(0))和 keccak256(0x12345678) == keccak256(uint32(0x12345678))。

在私链（private blockchain）上运行 sha256、ripemd160 或 ecrecover 可能会出现 out-of-gas 报错。因为私链实现了一种预编译合约，合约要在收到第一个消息后才会真正存在（虽然他们的合约代码是硬编码的）。而向一个不存在的合约发送消息，是导致 out-of-gas 问题的原因。另一种解决办法（work around）是，在你真正使用它们之前先发送 1 wei 到这些合约上来完成初始化。在官方和测试链上没有 out-of-gas 问题。

地址相关属性和函数

下面是地址相关的函数，在第 4 章地址类型中也有介绍，大家可以结合起来阅读。

- <address>.balance (uint256)：address 的余额，以 wei 为单位。
- <address>.transfer(uint256 amount)：发送给定数量的 Ether 到某个地址，以 wei 为单位。失败时抛出异常。
- <address>.send(uint256 amount) returns (bool)：发送给定数量的 Ether 到某个地址，以 wei 为单位，失败时返回 false。
- <address>.call(...) returns (bool)：发起底层的 call 调用，参数为函数选择器。失败时返回 false。
- <address>.callcode(...) returns (bool)：发起底层的 callcode 调用，参数为函数选择器，失败时返回 false。不鼓励使用，未来可能会移除。
- <address>.delegatecall(...) returns (bool)：发起底层的委托调用 delegatecall 调用，参数为函数选择器，失败时返回 false。由于委托调用的被调合约是在当前合约的环境下执行的，如果我们不知道被调合约具体做了什么事，委托调用将是一个很危险的操作。

下面的合约代码说明了地址属性如何使用，函数 balanceof()通过调用 <address>.balance 来获取当前合约的余额。

```
pragma solidity ^0.4.22;
contract testBalance {
    function  balanceOf() public constant returns (uint) {
        return this.balance; // this 表示当前合约
    }
}
```

注意：执行 send()函数有一些风险。如果调用栈的深度超过 1024 或 gas 耗光，交易都会失败。因此为了保证安全，必须检查 send()函数的返回值（失败时返回 false）。如果交易失败，会退回以太币。而 transfer()在失败时会抛出异常。实际上 addrA.transfer(addrB) 和 require(addrA.send(y))是等价的。因此，大部分时候，使用 transfer 函数是更好的选择。

尤其要注意的是 send()和 transfer()仅仅会分配到 2300 gas，如果接收地址是一个合约的话，则会触发回退函数的执行（第 8 章合约会进一步介绍回退函数）。如果回退函数复杂，则很容易使 send() 和 transfer() 执行失败。

下面是一个委托调用的例子：

```
pragma solidity ^0.4.22;
contract testBalance{
    function  balanceOf() public constant returns (uint) {
```

```
        return this.balance; // this 表示当前合约
    }
}

contract Mark {
    function Deposit() payable {}
    function call(address a) {   // 调用时需要传入 testBalance 合约的地址
        a.delegatecall(bytes4(keccak256 ("balanceOf()")));  //委托调用
    }
}
```

合约相关属性和函数

合约相关属性和函数有：

- this（当前合约的类型）：表示当前合约可以显式地转换为 Address。
- selfdestruct(address recipient)：销毁当前合约（也就是在第 2 章介绍的“自毁”），并把它的所有资金发送到给定的地址。
- suicide(address recipient)：selfdestruct 的别名。

另外，当前合约里的所有自定义的函数均可支持调用，包括当前函数本身。

来看一个例子，了解如何使用 selfdestruct()：

```
pragma solidity ^0.4.2;
contract Steal{
    address owner;
    function Steal() {
        owner = msg.sender;
}
function kill() {    // 这是销毁的标准做法
  if (msg.sender == owner) {
  selfdestruct(owner);
}
}
    function innocence() {
        selfdestruct(owner);  // 销毁合约
    }
}
```

```
contract Mark {
    function Deposit() payable {}
    function call(address a) {
        a.delegatecall(bytes4(sha3("innocence()")));
    }
}
```

在这段代码中，Steal 合约有一个 innocence 方法，这个函数虽然命名为“清白”，但如果 Mark 通过委托调用 innocence 方法，则 Mark 会把自己销毁掉，并且把存款发送到 Steal 合约的创建者那里。所以我们在委托调用其他合约的函数时，一定要特别小心。

本章小结

本章介绍了 Solidity 中的全局变量及函数，讲解了如何获取区块和交易的属性、如何通过 ABI 编码函数获取编码数据、如何处理错误函数，以及有关数学和加密的功能、与地址及合约相关的函数。

第 7 章

Solidity 表达式及控制结构

这一章介绍了 Solidity 中的表达式及控制结构。表达式是一种有值的语法结构，即一个表达式会产生一个值，它可以放在任何需要一个值的地方，比如一个变量本身就是一个表达式，函数调用及其参数、返回值都是表达式。

表达式和语句有一些区别，通常程序是由一系列语句组成的，不过某些需要语句的地方，可以使用一个表达式来代替。

表达式的求值顺序并不是确定的，但语句（statement）按顺序执行，布尔表达式的短路运算是可以保证的。本章主要内容有：

1. 函数参数
2. 控制结构
3. 函数调用表达式
4. 赋值表达式
5. 变量声明与作用范围
6. 错误处理

函数参数

与 JavaScript 一样，函数可以提供参数作为输入。与 JavaScript 和 C 不同的是，Solidity 还可以返回任意数量的参数作为输出。

输入参数

输入参数的声明方式与变量相同，未使用的参数可以省略变量名称。假设我们希望合约接受一种带有两个整数参数的外部调用，则可以这样写：

```
pragma solidity ^0.4.16;

contract Simple {
    function taker(uint _a, uint _b) public pure {
        // 使用 _a  _b.
    }
}
```

输出参数

输出参数的声明和输入参数一样，只不过它接在 returns 之后。假设我们希望返回两个结果，即两个给定整数的和、积，则可以这样写：

```
pragma solidity ^0.4.16;

contract Simple {
    function arithmetics(uint _a, uint _b)
        public
        pure
        returns (uint o_sum, uint o_product)
    {
        o_sum = _a + _b;
        o_product = _a * _b;
    }
}
```

可以省略输出参数的名称，也可以使用 return 语句指定输出值，return 可以返回多个值（见下节）。若返回一个没有赋值的参数，则默认为 0。

输入参数和输出参数可以在函数内的表达式中使用，也可以作为被赋值的对象（写在“=”的左边）。

控制结构

除 switch 和 goto 以外，JavaScript 中的大多数控制语句 Solidity 都支持。例

如，if、else、while、do、for、break、continue、return，并且语义均和 C 或 JavaScript 中的一样。

条件语句中的括号不能省略，但在单条语句前后的花括号{}可以省略。

下面是一个例子，可以简单说明各个控制语句的使用：

```
pragma solidity ^0.4.16;

contract controlTest {
    function testWhile() pure public returns (uint) {
        uint i = 0;
        uint sumOfOdd = 0;

        while (true) {
           i++;
           if (i % 2 == 0) {
                continue;
            }

            if (i>10) {
                break;
            }

            sumOfOdd += i;
        }
        return sumOfOdd;
    }


    function testfor() pure public returns (uint , uint) {
         uint sumOfOdd = 0;
         uint sumofEven = 0;
        for (uint i = 0; i < 10; i++) {
            if (i % 2 == 0) {
                sumofEven += i;
            } else {
                sumOfOdd += i;
            }
        }
```

```
        return (sumOfOdd, sumofEven);
    }
}
```

注意，在 Solidity 中没有像 C 和 JavaScript 那样从非布尔类型到布尔类型的转换，因此不能在条件语句中使用非布尔类型，所以 if(1){...}在 Solidity 中是不合法的。

返回多个值

当一个函数有多个输出参数时，可以使用 return (v0, v1, ..., vn) 语句，返回值的数量需要和输出参数声明的数量一致。如下：

```
function f() public pure returns (uint, bool, uint) {

        return (7, true, 2);

}

function callf() public {

        uint x;

        bool y;

        uint z;

        (x, y , z)  = f();

}
```

函数调用表达式

Solidity 中有两种函数调用方式，分别为内部函数调用和外部函数调用。下面我们分别讲解。

内部函数调用（Internal Function Call）

内部函数调用不会创建 EVM 消息调用。当前合约的函数可以直接调用，也可以递归调用，如下面这个例子：

```
pragma solidity ^0.4.16;

contract C {
    function g(uint a) public pure returns (uint ret) {
    return f(); } // 内部调用 f()
    function f() internal pure returns (uint ret) {
    return g(7) + f();// 内部调用 g() 及递归自身 f()
    }
}
```

这些函数调用被转换为 EVM 内部的简单指令“跳转”（jump）。这样带来的一个好处是，当前的内存不会被回收。在一个内部调用时传递一个内存型引用，效率将非常高。当然，仅仅是在同一个合约的函数之间才可以通过内部函数调用的方式进行调用。

外部函数调用（External Function Call）

外部调用会创建 EVM 消息调用。表达式的形式为：this.g(8);和 c.g(2);（这里的 c 是一个合约实例），通过 this 或使用一个合约实例来调用是外部调用函数的方式。它是一个消息调用，而不是 EVM 的指令跳转。

需要注意的是，在合约的构造器中不能使用 this 来调用函数，因为当前合约还没有创建完成。

其他合约的函数必须通过外部的方式调用。对于一个外部函数调用，所有函数的参数必须拷贝到内存中。

当调用其他合约的函数时，可以通过选项.value()和.gas()来分别指定要发送的以太币（以 wei 为单位）和 gas 值，如下所示：

```
pragma solidity ^0.4.0;

contract InfoFeed {
    function info() public payable returns (uint ret) {
    return 42;
    }
}

contract Consumer {
    InfoFeed feed;
    function setFeed(address addr) public {
```

```
    feed = InfoFeed(addr);
    }
    function callFeed() public {
    feed.info.value(10).gas(800)(); // 指定 value 和 gas 来调用 info 函数
    }
}
```

info()函数，必须使用 payable 关键字，否则不能通过 value()来接收以太币。

表达式 InfoFeed(addr)进行了一个显示的类型转换，表示给定的地址是合约 InfoFeed 类型，这里并不会执行构造器的初始化。在对显示的类型强制转换时是要非常小心的，不要调用一个我们不知道类型的合约函数。

我们也可以使用 function setFeed(InfoFeed _feed) { feed = _feed; }来直接进行赋值。注意 feed.info.value(10).gas(800)仅仅是对发送的以太币和 gas 值进行了设置，真正的调用是后面的括号()。调用 callFeed 时，需要预先存入一定量的以太币，否则会因余额不足报错。

> 如果我们不知道被调用的合约源代码，那么和它们交互会有潜在的风险。虽然被调用的合约继承自一个已知的父合约（继承仅仅要求正确实现接口，而不关注实现的内容），但是和他们交互就相当于把自己的控制权交给了被调用的合约，所以对方可以利用它做任何事。此外，被调用的合约可以改变调用合约的状态变量（state variable），在编写函数时我们要注意可重入性漏洞问题。

使用命名参数调用

函数调用的参数可以通过指定名称的方式调用，使用花括号{}括起来，参数顺序任意，但参数的类型和数量要与定义一致。如下所示：

```
pragma solidity ^0.4.0;

contract C {
    function f(uint key, uint value) public {
        // ...
    }

    function g() public {
```

```
        f({value: 2, key: 3});  // 命名参数
    }
}
```

省略函数参数名称

没有使用的参数名称可以省略（一般常见于返回值）。这些参数依然在栈（stack）上存在，但不可访问。

```
pragma solidity ^0.4.16;

contract C {
    // omitted name for parameter
    function func(uint k, uint) public pure returns(uint) {
        return k;
    }
}
```

赋值表达式

解构及返回多个值

Solidity 内置支持元组（tuple），它是由一个数量固定、类型可以不同的元素组成的一个列表。使用元组可以用来返回多个值，也可以用于同时赋值给多个变量（即为解构）。

> 解构的概念在函数式语言中较为常见。

我们看看以下示例：

```
pragma solidity ^0.4.16;

contract C {
    uint[] data;

    function f() public pure returns (uint, bool, uint) {
        return (7, true, 2);
    }
```

```
    function g() public {
        // 声明可赋值
        var (x, b, y) = f();
        // 赋值已声明过的变量
        (x, y) = (2, 7);
        // 交换变量值。只适用于值类型变量
        (x, y) = (y, x);

        // 支持省略一些元素，如果元组是以空元素为结尾，则其余部分可以省略
        (data.length,) = f();        // 设置长度为 7
        // 开头也可以省略
        (,data[3]) = f(); // 设置 data[3]为 2
        (x,) = (1,);    // 设置 x 为 1

        // 注意 (1,) 是一个一个元素的元组，(1) 只是 1。
    }
}
```

数组和自定义结构体的赋值

对于非值类型，比如数组和结构体，赋值的语法有一些复杂。赋值给一个状态变量时，会创建一份完全独立的拷贝。赋值给一个局部变量时，对于基本类型（如 2 个字节以内的静态类型）也会创建一份完全独立的拷贝。对于数据结构或者数组（包括 bytes 和 string）类型，局部变量则只是持有原始状态变量的一个引用。对这个局部变量再次赋值，并不会修改这个状态变量，而只是修改了引用。但修改这个本地引用变量的成员值，则会改变状态变量的值。

变量声明与作用范围

变量在声明后会用全零字节值作为默认值，也就是所有类型的默认值是典型的零态（zero-state）。举例来说，默认的 bool 的值为 false，uint 和 int 的默认值为 0。

对于从 byte1 到 byte32 定长的字节数组，每个元素都被初始化为对应类型的初始值（一个字节对应的是一个字节长的全零值，多个字节对应的是多个字节长的全零值）。对于变长的数组 bytes 和 string，默认值则为空数组和空字符串。

函数内定义的变量，其作用域是整个函数，不管它定义的位置。因为 Solidity 使用了 JavaScript 的变量作用域的规则。这与常规语言规定的从定义处开始到当前块结束不同。因此，下述代码编译时会抛出一个异常 Identifier already declared。

```
// 这个代码片段是无法编译的

pragma solidity ^0.4.16;

contract ScopingErrors {
    function scoping() public {
        uint i = 0;

        while (i++ < 1) {
            uint same1 = 0;
        }

        while (i++ < 2) {
            uint same1 = 0;  // 出错
        }
    }

    function minimalScoping() public {
        {
            uint same2 = 0;
        }

        {
            uint same2 = 0;  //出错
        }
    }

    function forLoopScoping() public {
        for (uint same3 = 0; same3 < 1; same3++) {
        }

        for (uint same3 = 0; same3 < 1; same3++)        }
    }
```

```
}
```

此外，如果一个变量被声明了，那么它会在函数开始前被初始化为默认值。所以下述例子是合法的。

```
pragma solidity ^0.4.0;

contract C {
    function foo() public pure returns (uint) {
        // baz 默认初始化为 0{
        // 出错
       uint bar = 5;
        if (true) {
            bar += baz;
        } else {
            uint baz = 10;    // 不会执行
        }
        return bar;  // returns 5
    }
}
```

错误处理

在上一章介绍全局变量的时候，我们介绍了 Solidity 中用来进行错误处理的 3 个函数，这一节我们将更深入地讲解错误处理。

什么是错误处理

错误处理是指在程序发生错误时的处理方式。Solidity 处理错误和我们常见的语言不一样，Solidity 是通过回退状态的方式来处理错误的（而绝大部分语言是通过捕获异常来处理错误的），发生异常时会撤销当前调用（及其所有子调用）所改变的状态，同时给调用者返回一个错误标识。注意**捕捉异常是不可能的**，因此没有 try ... catch...。

为什么 Solidity 要这样处理错误呢？我们回顾一下第 2 章介绍的事务概念。我们可以把区块链理解为全球共享的分布式事务性数据库。**全球共享**意味着参与这个网络的每一个人都可以读写其中的记录。如果想修改这个数据库中的内容，就必须创建一个事务。**事务**意味着要做的修改（假如我们想同时修改两个

值）只能被全部应用，只修改部分是不行的。Solidity 错误处理就是要保证每次调用都是事务性的。

如何处理

Solidity 提供了两个函数 assert 和 require 来进行条件检查。assert 函数通常用来检查（测试）内部错误，而 require 函数用来检查输入变量或合同状态变量是否满足条件，以及验证调用外部合约的返回值。另外，如果我们正确使用 assert，那么有一些 Solidity 分析工具（如实现中的 SMTChecker）就可以帮我们分析出智能合约中的错误。

除可以用两个函数 assert 和 require 来进行条件检查以外，还有两种方式可以触发异常：

1．revert 函数可以用来标记错误并回退当前调用，当前剩余的 gas 会返回给调用者。

2．使用 throw 关键字抛出异常（从 Solidity 0.4.13 版本开始，throw 关键字已被弃用）。

之前使用的 if(msg.sender != owner) { throw; }在功能上等价于下面的三个语句：

```
if(msg.sender != owner) { throw; }
assert(msg.sender == owner);
require(msg.sender == owner);
```

当子调用中发生异常时，异常会自动向上“冒泡”。不过也有一些例外，比如 send 和底层的函数调用 call、delegatecall、callcode，当发生异常时，这些函数返回 false。

注意：在一个不存在的地址上调用底层的函数 call、delegatecall、callcode 会成功返回，所以我们在进行调用时，应该优先进行函数存在性检查。

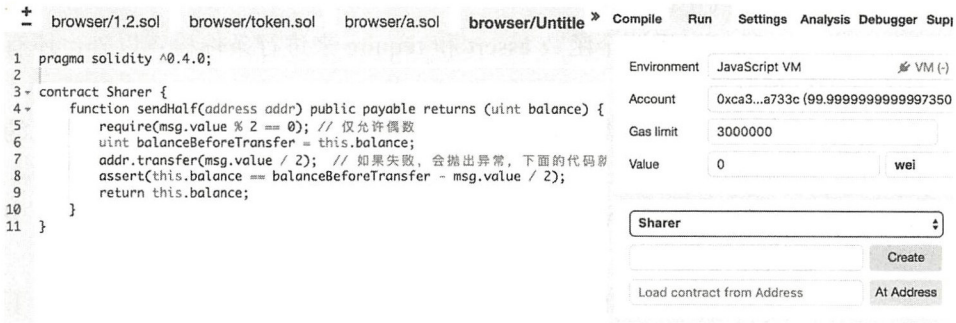
下面我们通过一个示例来说明如何使用 require 检查输入条件，以及使用 assert 进行内部错误检查：

```
pragma solidity ^0.4.0;

contract Sharer {
    function sendHalf(address addr) public payable returns (uint balance) {
        require(msg.value % 2 == 0); // 仅允许偶数
        uint balanceBeforeTransfer = this.balance;
```

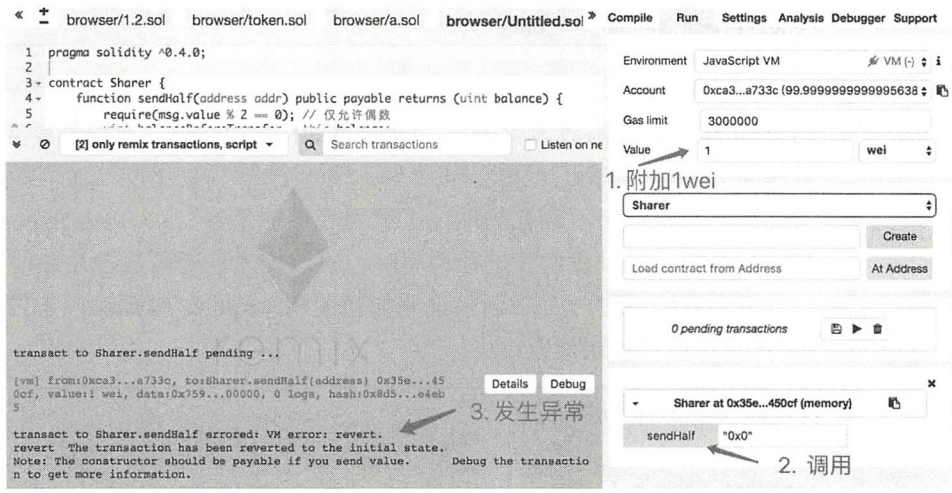
```
        addr.transfer(msg.value / 2);  // 如果失败，会抛出异常，下面的代码就不能
                                       // 执行了
        assert(this.balance == balanceBeforeTransfer - msg.value / 2);
        return this.balance;
    }
}
```

我们实际运行一下，看看异常是如何发生的。

首先打开 Remix，贴入代码，点击创建合约，如下图所示。



运行测试 1：附加 1wei（奇数）去调用 sendHalf，这时会发生异常，如下图所示。



运行测试 2：附加 2wei 去调用 sendHalf，运行正常。

运行测试 3：附加 2wei 及 sendHalf 参数为当前合约本身，在转账时发生异

常，因为合约无法接收转账，错误提示与上图类似。

assert 类型异常

在下述场景中自动产生 assert 类型的异常：

1．越界或负的序号值访问数组，如 i >= x.length 或 i < 0 时访问 x[i]。

2．序号越界或负的序号值访问一个定长的 bytesN。

3．被除数为 0，如 5/0 或 23% 0。

4．对一个二进制数移动一个负的值，如 5<<i; i 为–1。

5．整数进行显式转换为枚举时，将过大值、负值转为枚举类型并抛出异常。

6．调用未初始化内部函数类型（参考函数类型章节）的变量。

7．调用 assert 的参数为 false。

require 类型异常

在下述场景中自动产生 require 类型的异常：

1．调用 throw。

2．调用 require 的参数为 false。

3．通过消息调用一个函数，但在调用的过程中并没有正确结束（例如 gas 不足没有匹配到对应的函数，或是被调用的函数出现异常）。底层操作如 call、send、delegatecall 或 callcode 除外，它们不会抛出异常，但它们会通过返回 false 来表示失败。

4．在使用 new 创建一个新合约时因为第 3 条的原因而没有正常完成。

5．调用外部函数时，被调用的对象不包含代码。

6．合约没有 payable 修饰符的 public 的函数在接收以太币时（包括构造函数和回退函数）会出现异常。

7．合约通过一个 public 的 getter 函数接收以太币。

8．.transfer()执行失败。

assert 与 require 的详细对比

两种类型的相同点是异常都会撤销所有操作，这是为了保证交易的原子性（即一致性，要么全部执行，要么一点都不执行，不能只改变一部分）。

两种类型的不同点，如下：

1．gas 消耗不同

assert 类型的异常会消耗掉所有剩余的 gas，而 require 不会消耗剩余 gas（剩

余 gas 会返还给调用者）。

2．操作符不同

当发生 require 类型的异常时，Solidity 会执行一个回退操作（REVERT 指令 0xfd）。

当发生 assert 类型的异常时，Solidity 会执行一个无效操作（无效指令 0xfe）。

该使用哪一个？

下面是使用 require()的一些经验总结：

1．用于检查用户输入。

2．用于检查合约调用返回值，如 require(external.send(amount))。

3．用于检查状态，如 msg.send == owner。

4．通常用于函数的开头。

5．不知道使用哪一个的时候，就使用 require。

下面是使用 assert()的一些经验总结：

1．用于检查溢出错误，如“z = x + y ; assert(z >= x);”。

2．用于检查不应该发生的异常情况。

3．用于在状态改变之后，检查合约状态。

4．尽量少使用 assert;。

5．通常用于函数中间或结尾。

本章小结

在本章中，我们详细介绍了几种常见的表达式形式，如函数参数、函数调用、赋值及变量声明和作用范围；控制结构部分介绍了函数返回值及错误处理。

第 8 章 合约

本章将详细介绍合约，主要内容包含：

1. 合约概述
2. 创建合约
3. 可见性分析
4. 访问函数
5. 函数修改器
6. 状态常量
7. 视图函数
8. 纯函数
9. 回退函数
10. 函数重载
11. 事件
12. 继承
13. 构造函数
14. 抽象合约
15. 接口
16. 库
17. Using for 指令

合约概述

Solidity 中合约和面向对象语言中的类差不多。合约包含状态变量（状态变量的数据保存在链上的区块中）及函数。调用另一个合约实例的函数时，会执行一个 EVM 函数调用，这个操作会切换执行时的上下文，这时上一个合约的状态变量就无法访问了。

创建合约

合约可以通过发起交易（通过 Web3 发起）或是 Solidity 创建。

在以太坊上动态地创建合约可以使用 JavaScript API Web3.js（在第 13 章，我们会进一步介绍 Web3.js）的 web3.eth.Contract 来创建合约。当创建合约时，合约的构造函数（使用 constructor 定义的函数）会被调用，用于初始化。构造器函数是可选的，但仅能有一个构造函数，因此构造函数不支持重载。

如果一个合约想创建另一个合约，那么必须要提前知道源码（被创建合约的代码必须是已知的），因此不支持嵌套创建。

下面是使用 Web3.js API 来创建合约的代码：

```
var source = "contract A { function A(uint a) {} }";
// 下面的代码可以使用编译器，如 remix 生成

var a = 10;
var aContract = web3.eth.contract([{"inputs":[{"name":"a","type":"
uint256"}],"payable":false,"stateMutability":"nonpayable","type":"constructo
r"}]);
var a = aContract.new(
   a,
   {
     from: web3.eth.accounts[0],
     data:
'0x60606040523415600e57600080fd5b604051602080606c833981016040528080519060200
190919050505060358060376000396000f300606060404052600080fd00a165627a7a723058207
bb3515740fd13f73ddf67e9a7be34568648fff52fc7e50a3fd7af0df72d34730029',
     gas: '4700000'
   }, function (e, contract){
    console.log(e, contract);
```

```
    if (typeof contract.address !== 'undefined') {
        console.log('Contract mined! address: ' + contract.address + '
transactionHash: ' + contract.transactionHash);
    }
 })
```

使用 new 创建合约

合约内可以通过 new 关键字来创建一个新合约。

```
pragma solidity ^0.4.0;

contract D {
    uint x;
    function D(uint a) public payable {
        x = a;
    }
}

contract C {
    D d = new D(4);          // 将作为C构造的一部分被执行

    function createD(uint arg) public {
        D newD = new D(arg);
    }

    function createAndEndowD(uint arg, uint amount) public payable {
        // 在创建合约时发送以太币
        D newD = (new D).value(amount)(arg);
    }
}
```

可以在创建的合约中发送 Ether，但不能限制 gas。如果创建发生 out-of-stack 或无足够的余额，则会抛出一个异常。

再看一个使用 Solidity 创建合约的代码，如下所示。

```
pragma solidity ^0.4.16;

contract OwnedToken {
```

```
    // TokenCreator 是在下面定义一个合约类型
    TokenCreator creator;
    address owner;
    bytes32 name;


    constructor (bytes32 _name) public {
        owner = msg.sender;
        //地址转为 TokenCreator 合约
        creator = TokenCreator(msg.sender);
        name = _name;
    }

    function changeName(bytes32 newName) public {
        //只有创建者能改名
        if (msg.sender == address(creator))
            name = newName;
    }

    function transfer(address newOwner) public {
        // 只有 owner 能转移 token
        if (msg.sender != owner) return;

        if (creator.isTokenTransferOK(owner, newOwner))
            owner = newOwner;
    }
}

// 用来创建合约
contract TokenCreator {
    function createToken(bytes32 name)
       public
       returns (OwnedToken tokenAddress)
    {
        // 创建 token 合约，返回合约地址
        return new OwnedToken(name);
    }
```

```
    function changeName(OwnedToken tokenAddress, bytes32 name)  public {
        tokenAddress.changeName(name);
    }

    function isTokenTransferOK(address currentOwner, address newOwner) public
        view
        returns (bool ok)
    {
       address tokenAddress = msg.sender;
        return (keccak256(newOwner) & 0xff) == (bytes20(tokenAddress) & 0xff);
    }
}
```

可见性

Solidity 有两种函数调用方式（在第 7 章函数调用表达式中介绍过），一种是内部调用，不会创建 EVM 调用（也叫作消息调用），另一种是外部调用，会创建 EVM 调用（即会发起消息调用）。Solidity 对函数和状态变量提供了四种可见性，分别是 external、public、internal、private。其中函数默认的可见性是 public。状态变量默认的可见性是 internal。

external

外部函数是合约接口的一部分，所以我们可以从其他合约或通过交易来发起调用。而一个外部函数 f 不能通过内部的方式来发起调用，例如：不可以使用 f()发起调用，只能使用 this.f()发起调用。外部函数在接收大的数组数据时更加有效。

public

公开函数也是合约接口的一部分，可以同时支持内部调用及消息调用。对于 public 类型的状态变量，会自动创建一个访问器，这是一个与状态变量名字相同的函数，用来获取状态变量的值。

internal

这样声明的函数和状态变量只能在内部访问，比如在当前合约中调用，或者在继承的合约里访问。需要注意的是访问时不能加前缀 this，前缀 this 表示的

是通过外部方式访问。

private

私有函数和状态变量仅在当前合约中可以访问，在继承的合约内不可访问。

> 备注：所有在合约内的东西对外部的观察者来说都是可见的，将某些东西标记为 private 仅仅阻止了其他合约来进行访问和修改，但并不能阻止其他人看到相关的信息。

可见性的标识符的定义位置，通常放在状态变量类型的后面，函数的话是在参数列表和返回关键字中间。来看一个定义的例子：

```
pragma solidity ^0.4.16;

contract C {
    function f(uint a) private returns (uint b) { return a + 1; }
    function setData(uint a) internal { data = a; }
    uint public data;
}
```

在下面的例子中，D 可以调用 c.getData()来访问 data 的值，但不能调用 f。合约 E 继承自 C，所以它可以访问 compute 函数。

```
pragma solidity ^0.4.0;

contract C {
    uint private data;

    function f(uint a) private returns(uint b) { return a + 1; }
    function setData(uint a) { data = a; }
    function getData() public returns(uint) { return data; }
    function compute(uint a, uint b) internal returns (uint) { return a+b; }
}


contract D {
    function readData() {
```

```
        C c = new C();
        uint local = c.f(7); // error: member "f" is not visible
        c.setData(3);
        local = c.getData();
        local = c.compute(3, 5); // error: member "compute" is not visible
    }
}


contract E is C {
    function g() {
        C c = new C();
        uint val = compute(3, 5);  // acces to internal member (from derivated
to parent contract)
    }
}
```

访问函数（Getter Function）

编译器会自动为所有的 public 的状态变量生成访问函数。下面的合约例子中，编译器会生成一个名叫“data”的无参数的函数，返回值是 uint 类型的值 data。状态变量的初始化可以在定义时完成：

```
pragma solidity ^0.4.0;

contract C {
    uint public data = 42;
}

contract Caller {
    C c = new C();
    function f() public {
        uint local = c.data();
    }
}
```

访问函数有外部可见性。如果是内部访问的，则可以直接访问变量（不用

this)。但如果使用外部方式来访问(通过 this.),则必须通过函数的方式来调用。如下所示:

```
pragma solidity ^0.4.0;

contract C {
    uint public data;
    function x() public {
        data = 3; // internal access
        uint val = this.data(); // external access
    }
}
```

来一个复杂一点的例子:

```
pragma solidity ^0.4.0;

contract Complex {
    struct Data {
        uint a;
        bytes3 b;
        mapping (uint => uint) map;
    }
    mapping (uint => mapping(bool => Data[])) public data;
}
```

编译器会生成以下函数:

```
function data(uint arg1, bool arg2, uint arg3) public returns (uint a, bytes3
b) {
    a = data[arg1][arg2][arg3].a;
    b = data[arg1][arg2][arg3].b;
}
```

注意,结构体里的 mapping 初始化被省略了,因为没有一个很好的方式来提供键值。

函数修改器（Function Modifier）

我们在第 8 章合约中，提到过函数修改器。函数修改器可以用来改变一个函数的行为，比如用于在函数执行前检查某种前置条件。

> 熟悉 Python 的读者会发现函数修改器的作用和 Python 的装饰器很相似。

修改器是一种可被继承合约的属性，同时还可被继承合约重写（Override）。下面我们来看一段示例代码：

```
pragma solidity ^0.4.11;

contract owned {
    function owned() public { owner = msg.sender; }
    address owner;

    // 定义了一个函数修改器，可被继承
    // 修改时，函数体被插入到 "_;" 处
    // 不符合条件时，将抛出异常

    modifier onlyOwner {
        require(msg.sender == owner);
        _;
    }
}

contract mortal is owned {
    //  使用继承的 onlyOwner
    function close() public onlyOwner {
        selfdestruct(owner);
    }
}

contract priced {
    // 函数修改器可接收参数
    modifier costs(uint price) {
        if (msg.value >= price) {
```

```
            _;
        }
    }
}

contract Register is priced, owned {
    mapping (address => bool) registeredAddresses;
    uint price;

    function Register(uint initialPrice) public { price = initialPrice; }

    // 需要提供 payable 以接受以太
    function register() public payable costs(price) {
        registeredAddresses[msg.sender] = true;
    }

    function changePrice(uint _price) public onlyOwner {
        price = _price;
    }
}
```

上面的 onlyOwner 就是一个函数修改器。当用这个修改器修饰一个函数时，函数必须满足 onlyOwner 的条件才能运行。这里的条件是：函数必须是合约创建的才能调用，否则抛出异常。

多个函数修改器

如果同一个函数有多个修改器，并且他们之间以空格隔开，那么修改器会依次检查执行。

在修改器中或函数内的显式的return语句，仅仅跳出当前的修改器或函数。返回的变量会被赋值，但执行流会在前一个修改器后面定义的“_”后继续执行，如下所示：

```
contract Mutex {
    bool locked;
    modifier noReentrancy() {
        require(!locked);
        locked = true;
```


```
        _;
        locked = false;
    }

    // 防止递归调用
    // return 7 之后，locked = false 依然会执行
    function f() public noReentrancy returns (uint) {
        require(msg.sender.call());
        return 7;
    }
}
```

修改器的参数可以是任意表达式。在此上下文中，所有的函数引入的符号在修改器中均可见。但修改器中引入的符号在函数中不可见，因为它们有可能被重写。

理解修改器的执行次序

再来看一个复杂一点的例子，让我们深入理解修改器：

```
pragma solidity ^0.4.11;


contract modifysample {

    uint a = 10;

    modifier mf1 (uint b) {
        uint c = b;
        _;
        c = a;
        a = 11;
    }

     modifier mf2 () {
        uint c = a;
        _;
    }
```

```
    modifier mf3() {
        a = 12;
        return ;
        _;
        a = 13;
    }

    function test1() mf1(a) mf2 mf3 public   {
        a = 1;
    }

     function test2() public constant returns (uint)   {
        return a;
    }
}
```

上面的智能合约在运行 test1()之后，状态变量 a 的值是多少？是 1、11、12 还是 13 呢？答案是 11，大家可以运行 test2 获取 a 值。

我们来分析一下 test1。它扩展之后是这样的：

```
uint c = b;
        uint c = a;
            a = 12;
            return ;
            _;
            a = 13;
c = a;
a = 11;
```

这个时候就一目了然了，最后 a 为 11。

状态常量

状态常量可以被定义为 constant，它在编译期间通过一个表达式赋值，不过表达式有一些限制：

1. 不允许访问 storage。
2. 不允许访问区块链数据，如 now、this.balance、block.number。

3．不允许访问合约执行中的数据，如 msg.value 或 gasleft()。

4．不允许向外部合约发起调用。

但内置的函数 keccak256、keccak256、ripemd160、ecrecover、addmod、mulmod 可以允许调用，即使它们调用的是外部合约。

编译器并不会为常量在 storage 上预留空间，每个使用的常量都会被对应的常量表达式所替换（也许优化器会直接替换为常量表达式的结果值）。

目前不是所有的类型都支持常量，现在支持的仅有值类型和字符串。

```
pragma solidity ^0.4.0;

contract C {
    uint constant x = 32**22 + 8;
    string constant text = "abc";
    bytes32 constant myHash = keccak256("abc");
}
```

视图函数（View Function）

函数可以声明为 view，表示它不能修改状态。下面几种情况被认为是修改了状态。

1．写状态变量。

2．触发事件。

3．创建其他的合约。

4．call 调用附加了以太币。

5．调用了任何没有 view 或 pure 修饰的函数。

6．使用了低级别的调用（low-level call）。

7．使用了包含特定操作符的内联汇编。

看一个例子：

```
pragma solidity ^0.4.16;

contract C {
    function f(uint a, uint b) public view returns (uint) {
        return a * (b + 42) + now;
    }
```


```
}
```

有几个地方需要注意一下：

1．声明为 view 和声明为 constant 是等价的，constant 是 view 的别名。constant 计划在 Solidity 0.5.0 版本之后被弃用。

2．访问函数都被标记为 view。

当前编译器并未强制要求声明为 view，但建议大家对于不会修改状态的函数标记为 view。

纯函数（Pure Function）

函数可以声明为 view，表示它既不能读取状态，也不能修改状态。除上一节介绍的几种修改状态的情况以外，以下几种情况被认为是读取了状态：

1．读状态变量。

2．访问了 this.balance 或 <address>.balance。

3．访问了 block, tx, msg 的成员（msg.sig 和 msg.data 除外）。

4．调用了任何没有 pure 修饰的函数。

5．使用了包含特定操作符的内联汇编。

看一个例子：

```
pragma solidity ^0.4.16;

contract C {
    function f(uint a, uint b) public pure returns (uint) {
        return a * (b + 42);
    }
}
```

回退函数（Fallback Function）

一个合约可以有一个（最多也只能有一个）没有名字的函数。这个函数无参数，也无返回值。这个函数称为回退函数，在以下几种情况下被调用：

如果调用合约时，没有匹配上任何一个函数（或者没有提供数据），则会调用回退函数。

当给合约转以太币时（没有任何其他数据），回退函数也会被执行，并且需

要标记为 payable。如果没有回退函数，就无法接收以太币转账。

有一个情况需要注意：有时转账者仅仅会提供 2300 gas 来执行，以保证回退函数尽量使用较少的 gas，以下操作的消耗会大于 2300 gas。

1．写存储（storage）。

2．创建一个合约。

3．执行一个外部函数调用，会花费非常多的 gas。

4．发送 Ether。

回退函数也像其他函数一样，只要有足够多的 gas 就可以执行任何复杂的操作。

另外，尽管回退函数没有参数，其依然可以使用 msg.data 读取调用时提供的 payload。

注意：

一个没有定义回退函数的合约如果接收 Ether，那么会触发异常并返还 Ether（从 Solidity 0.4.0 开始）。所以合约要接收 Ether，必须实现回退函数。但是，如果合约地址作为挖矿交易的接收者或 selfdestruct 的接收者，则可以没有回退函数。

来看一个例子：

```
pragma solidity ^0.4.0;

contract Test {

    // 由于没有 payable ，向这个合约转账时会发生异常
    function() public { x = 1; }
    uint x;
}


// 保存所有接收的以太币，无法拿回
contract Sink {
    function() public payable { }
}

contract Caller {
    function callTest(Test test) public {
        test.call(0xabcdef01); // 不存在的 hash
```


```
        // results in test.x becoming == 1.

        // 无法编译，通过其他方式发送则失败
        //test.send(2 ether);
    }
}
```

函数重载（Function Overloading）

函数重载是指一个合约可以有多个参数不同的同名函数，如下所示。

```
pragma solidity ^0.4.16;

contract A {
    function f(uint _in) public pure returns (uint out) {
        out = 1;
    }

    function f(uint _in, bytes32 _key) public pure returns (uint out) {
        out = 2;
    }
}
```

有一种情况需要注意，如果仅仅是Solidity类型不一样，但在外部接口（ABI）上表现一样的话，那么是无法函数重载的。例如，下面的例子就是无法编译的。

```
pragma solidity ^0.4.16;

contract A {
    function f(B _in) public pure returns (B out) {
        out = _in;
    }

    function f(address _in) public pure returns (address out) {
        out = _in;
    }
}
```


```
contract B {
}
```

重载的参数匹配问题

重载函数在调用时，会根据调用提供的参数的类型去匹配重载函数，这期间可能会发生隐式的类型转换。如果可以匹配上多个重载函数，则会调用失败。来看一个例子：

```
pragma solidity ^0.4.16;

contract A {
    function f(uint8 _in) public pure returns (uint8 out) {
        out = _in;
    }

    function f(uint256 _in) public pure returns (uint256 out) {
        out = _in;
    }
}
```

f(50)会失败，因为 50 可以隐式转换为 uint8 和 uint256。而 f(256)则会调用 f(uint256)，因为 256 不能转换为 uint8。

注意：返回值不参与匹配。

事件

我们在合约一章中提到过：事件是以太坊虚拟机（EVM）日志基础设施提供的一个便利接口。在 DApp 的应用中，如果监听了某个事件，那么当事件发生时则会进行回调。

事件在合约中可以被继承。当被发送事件（调用）时，会触发参数存储到交易的日志中（一种区块链上的特殊数据结构）。这些日志与合约的地址关联，并记录到区块链中，只要区块可以访问就一直存在（至少 Frontier、Homestead 是这样的，Serenity 也许会更改）。日志和事件在合约内是无法被访问的，即使是创建了日志的合约也不行。

日志的 SPV（简单支付验证）证明是可能的，如果一个外部的实体提供了

一个证明合约，那么它可以证明日志在区块链中是否存在。但需要注意的是，由于合约中仅能访问最近的 256 个区块哈希，所以还需要提供区块头信息。

事件是通过关键字 event 来声明的，event 不需要实现（可以认为它是一个用来被监听的接口）。

下面是一个简单的例子：

```
pragma solidity ^0.4.0;

contract ClientReceipt {
    event Deposit(
        address indexed _from,
        bytes32 indexed _id,
        uint _value
    );

    function deposit(bytes32 _id) public payable {
        // 使用 event 来触发事件
        // 任何对该函数的调用都可以通过 JavaScript API 监听到
        emit Deposit(msg.sender, _id, msg.value);
    }
}
```

下面是使用 JavaScript Web3 API（在第 13 章会提供一个完整的示例）来获取日志的例子。

```
var abi = /* 编译器生成的 abi */;
var ClientReceipt = web3.eth.contract(abi);
var clientReceipt = ClientReceipt.at("0x1234...ab67" /* 合约地址 */);

var event = clientReceipt.Deposit();

// 监听更改
event.watch(function(error, result){
    // result 会包含除参数之外的一些其他信息
    if (!error)
        console.log(result);
});
```


```
// 或是通过传一个回调函数来监听
var event = clientReceipt.Deposit(function(error, result) {
    if (!error)
        console.log(result);
});
```

事件主题（Topic）

主题是用来把事件索引化（Index）的数值。没有主题，就不能搜索事件（主题的作用是用来检索事件的）。一个事件最多可以有四个主题。

第一个主题是事件签名，剩下三个主题是索引化的参数数值（即最多有三个参数接收属性可以被设置为索引）。设置为主题后，可以允许通过这个参数来查找日志，甚至可以按特定的值过滤。如果参数是字符串、字节或者数组，那么主题是它的 keccak-256 哈希。

注意：1. 事件签名 hash 是其中一个主题，匿名事件除外，这意味着对于匿名事件无法通过名字来过滤。2. 被索引的参数将不会保存它们自己，可以搜索它们的值，但不能检索值本身。所有未被索引的参数将被作为日志的一部分被保存起来。

下面通过一个例子来帮助大家理解主题，假如有如下事件：

```
event PersonCreated(uint indexed age, uint indexed height);

// 通过参数触发
emit PersonCreated(26, 176);
```

这里会生成 3 个主题：

- 0x6be15e8568869b1e100750dd5079151b32637268ec08d199b318b793181b8a7d，它是事件的签名，计算方法是：Keccak-256(“PersonCreated(uint256, uint256)”)。
- 0x36383cc9cfbf1dc87c78c2529ae2fcd4e3fc4e575e154b357ae3a8b2739113cf，它是年龄 26 的 Keccak-256 值。
- 0x048dd4d5794e69cea63353d940276ad61f89c65942226a2bb5bd352536892f82，它是身高 176 的 Keccak-256 值。

在节点上就会创建这样的可搜索的索引，之后可以在 Web3 中对索引进行过滤搜索，比如可以过滤出所有 26 岁的人，只需要使用以下代码：

```
var createdEvent = myContract.PersonCreated({age: 26});
createdEvent.watch(function(err, result) {
       if (err) {
       console.log(err)
       return;
       }
       console.log("Found ", result);
})
```

底层的日志接口（Low-level Interface to Log）

通过函数 log0、log1、log2、log3、log4 直接访问底层的日志。logi 表示带 i + 1 个 bytes32 类型的参数，i 表示的就是可带参数的数目，从 0 开始计数。

其中第一个参数会被用来作为日志的数据部分，其他的参数作为主题。前面例子中的事件可进行以下更改：

```
pragma solidity ^0.4.10;

contract C {
    function f() public payable {
        bytes32 _id = 0x420042;
        log3(
            bytes32(msg.value),
        bytes32(0x50cb9fe53daa9737b786ab3646f04d0150dc50ef4e75f59509d83667ad5
adb20),
            bytes32(msg.sender),
            _id
        );
    }
}
```

其中的十六进制字符串是事件的签名，计算方式是 keccak256("Deposit (address, hash256,uint256)")。

继承

Solidity 继承使用的是关键字 is（类似于 Java 等语言的 extends 或 implements）。Solidity 是通过复制包括多态的代码来支持多重继承的。

如果没有明确指定调用哪一个合约的函数，那么最终被派生的方法通常会被调用。

如果一个合约从多个其他合约那里继承，那么在区块链上仅会创建一个合约，并且在父合约里的代码会被复制用来创建被继承合约。

这个继承方式跟 Python 有些类似，特别是在处理多继承方面。

下面我们用一个例子来详细说明：

```
pragma solidity ^0.4.16;

contract owned {
    function owned() { owner = msg.sender; }
    address owner;
}

// is 表示继承。继承的合约可以访问所有非 private 成员（包括内部函数及状态变量）
contract mortal is owned {
    function kill() {
        if (msg.sender == owner) selfdestruct(owner);
    }
}

// 这是一个抽象合约，仅提供了一个接口函数（没有函数体），这样的合约仅能当接口使用
contract Config {
    function lookup(uint id) public returns (address adr);
}

contract NameReg {
    function register(bytes32 name) public;
    function unregister() public;
 }

// 多继承，注意 mortal 的基类也是 owned
contract named is owned, mortal {
    function named(bytes32 name) {
        Config config = Config(0xD5f9D8D94886E70b06E474c3fB14Fd43E2f23970);
        NameReg(config.lookup(1)).register(name);
    }
```

```
    // 函数可以复写，需要保持参数与类型和父类一致，否则会报错
    function kill() public {
        if (msg.sender == owner) {
            Config config =
Config(0xD5f9D8D94886E70b06E474c3fB14Fd43E2f23970);
            NameReg(config.lookup(1)).unregister();
            // 可以调用指定的函数
            mortal.kill();
        }
    }
}

// 如果构造函数有参数，则需要在继承时提供参数
contract PriceFeed is owned, mortal, named("GoldFeed") {
   function updateInfo(uint newInfo) public {
      if (msg.sender == owner) info = newInfo;
   }

   function get() public view returns(uint r) { return info; }

   uint info;
}
```

注意，上面例子的 kill()方法中，我们调用了 motal.kill()，调用了父合约的销毁函数（destruction）。但这可能会引发一些问题，看下面的例子：

```
pragma solidity ^0.4.0;

contract owned {
    function owned() public { owner = msg.sender; }
    address owner;
}

contract mortal is owned {
    function kill() public {
        if (msg.sender == owner) selfdestruct(owner);
    }
```

```
}

contract Base1 is mortal {
    function kill() public { mortal.kill(); }
}

contract Base2 is mortal {
    function kill() public {  mortal.kill(); }
}

contract Final is Base1, Base2 {
}
```

对 Final.kill()的调用只会调用 Base2.kill()（最后重载的函数），而派生重写会跳过Base1.kill，因为它根本就不知道有Base1。一个变通的方法就是使用super，看下面的例子：

```
pragma solidity ^0.4.0;

contract owned {
    function owned() public { owner = msg.sender; }
    address owner;
}

contract mortal is owned {
    function kill() public {
        if (msg.sender == owner) selfdestruct(owner);
    }
}

contract Base1 is mortal {
    function kill() public {  super.kill(); }
}


contract Base2 is mortal {
    function kill() public { super.kill(); }
}
```

```
contract Final is Base1, Base2 {
}
```

如果 Base2 调用了函数 super，那么它不仅会调用基类的合约函数，还会调用继承关系图谱上的下一个基类合约，所以会调用 Base1.kill()。需要注意的是最终的继承图谱将会是 Final、Base2、Base1、mortal、owned。使用 super 时调用的实际函数在使用它的类的上下文中是未知的，尽管它的类型是已知的。这类似于普通虚函数查找。

构造函数（Constructor）

构造函数也叫“构造器”，通常用来完成合约的初始化变量赋值。它是一个可选函数，构造函数使用关键字 constructor 表示。它仅在创建合约时执行一次。如果没有实现构造函数，那么合约会添加一个默认的构造函数。

```
constructor() public {}
```

构造函数要么是 public，要么是 internal。如果是 internal，则作为抽象合约。

来看一个例子：

```
pragma solidity ^0.4.22;

contract A {
    uint public a;

    constructor (uint _a) internal {
        a = _a;
    }
}

contract B is A(1) {
    constructor () public {}
}
```

在 Solidity 0.4.22 版本之前，构造是使用与合约同名的函数来实现的，语法如下所示（不过现在这个写法已经弃用了）。

```
pragma solidity ^0.4.11;

contract A {
    uint public a;

    function A(uint _a) internal {
        a = _a;
    }
}

contract B is A(1) {
    function B() public {}
}
```

父合约构造函数的参数

派生的合约需要调用所有父合约的构造函数，并且需要提供所有父合约需要的参数。有两种方式来提供，如下所示。

```
pragma solidity ^0.4.22;

contract Base {
    uint x;
    constructor (uint _x) public { x = _x; }
}

contract Derived1 is Base(7) {
    constructor(uint _y) public {
    }
}

contract Derived2 is Base {
    constructor(uint _y) Base(_y * _y) public {
    }
}
```

一种称为“继承列表”式，即直接在继承列表中使用 is Base(7)，示例代码 Derived1 合约就是使用这个方式。

另一种称为“修改器风格”式，示例代码 Derived2 合约就是使用这个方式。Base(_y * _y)就像是一个函数修改器，它会作为修饰派生构造函数的一部分得到执行。

第一种方式对于构造器是常量的情况比较方便，可以直接说明合约的行为。第二种方式适用于构造的参数值是由派生合约指定的情况。

注意，不应该同时使用这两种方式。另外，如果继承合约没有指定父合约的构造参数，则合约会被当成一个抽象合约（后面章节会介绍抽象合约）。

多继承与线性化

支持多继承的编程语言需要解决几个问题，其中之一就是菱形继承问题又称“钻石问题”。Solidity 的解决方案可以参考 Python，使用 C3 线性化方式来强制将基类合约转换为一个有向无环图（DAG）的特定顺序。基类合约在 is 后的顺序变得非常重要。例如，下面的日志就无法通过编译：

```
pragma solidity ^0.4.0;

contract X {}
contract A is X {}
contract C is A, X {}
```

原因是 C 会请求 X 来重写 A（因为继承定义的顺序是 A、X），但 A 自身又是重写 X 的，所以这是一个不可解决的矛盾。

一个简单的指定基类合约的继承顺序，原则上是从**最接近基类**到**最接近派生类的**。

同名问题

如果在继承时出现一个合约同时存在多个相同名字的修改器或函数，那么会产生错误。如果事件与修改器重名或函数与事件重名，则都会产生错误。

例外的是，状态变量的 getter 可以覆盖一个 public 函数。

```
// 下面的代码无法通过编译，因为函数和事件同名了
pragma solidity ^0.4.22;

contract test {

    event Test(uint a);
```

```
    function Test(uint a) {
    }

}
```

抽象合约（Abstract Contract）

抽象合约和其他语言里的抽象类是类似的，它是一个合约中没有函数体（实现）的函数（函数直接以;结尾）的合约。下面就是一个抽象合约。

```
pragma solidity ^0.4.0;

contract Feline {
    function utterance() public returns (bytes32);
}
```

这样的合约即使在合约内包含一些正常的函数，也不能部署运行。它们只能作为基合约被继承。例如：

```
pragma solidity ^0.4.0;

contract Feline {
    function utterance() public returns (bytes32);
}

contract Cat is Feline {
    function utterance() public returns (bytes32) { return "miaow"; }
}
```

如果一个合约从一个抽象合约里继承，但却没实现所有函数，那么它也是一个抽象合约。注意一下，没有实现的函数和第 4 章讲的函数类型在语法上很像，但他们不是一回事儿。

下面是一个没有实现的函数（一个函数声明）：

```
function foo(address) external returns (address);
```

下面是一个函数类型的例子，声明一个变量。变量的类型是函数：

```
function(address) external returns (address) foo;
```

抽象合约分离了定义和实现，但也提供了扩展的能力。

接口（Interface）

接口和抽象合约类似，与之不同的是，接口内没有任何函数是已实现的，同时还有以下限制：

1. 不能继承其他合约或接口。
2. 不能定义构造器。
3. 不能定义变量。
4. 不能定义结构体。
5. 不能定义枚举类。

其中的一些限制可能在未来放开。

接口被限制为合约 ABI（ABI 将在第 10 章讲解）定义可以表示的内容，ABI 和接口定义之间应该可以进行转换而不会有任何信息丢失。接口声明用自己的关键词 interface 表示：

```
pragma solidity ^0.4.11;

interface Token {
    function transfer(address recipient, uint amount) public;
}
```

合约可以继承接口，就像合约可以继承其他的合约一样。

库

库与合约类似，它也部署在一个指定的地址上（仅被部署一次，当然代码可以在不同的合约反复使用），然后通过 EVM 的特性 DELEGATECALL（Homestead 之前是用 CALLCODE）来复用代码。库函数在被调用时，库代码是在发起合约（下文称**主调合约**：主动发起 DELEGATECALL 调用的合约）的上下文中执行的，使用 this 将会指向主调合约，而且库代码可以访问主调合约的存储（storage）。

因为库合约是一个独立的代码，所以它仅可以访问主调合约明确提供的状

态变量。

对比普通合约来说，库有以下的限制（这些限制也可能在未来的版本中解除）：

1. 无状态变量。

2. 不能继承或被继承。

3. 不能接收以太币。

4. 不能销毁一个库。

不会修改状态变量（例如被声明 view 或 pure），库函数只能通过直接调用（不用 DELEGATECALL）。这是因为库被认为是与状态无关的。

库有许多使用场景。其中两个主要的场景如下：

1. 如果有许多合约，它们有一些共同代码，则可以把共同代码部署成一个库。这将节省 gas，因为 gas 也依赖于合约的规模。因此，可以把库想象成使用其合约的父合约。使用父合约（而非库）切分共同代码不会节省 gas，因为在 Solidity 中，继承通过复制代码工作。

2. 库可用于给数据类型添加成员函数。（参见下一节 Using for。）

由于库被当成隐式的父合约（它们不会显式地出现在继承关系中，但调用库函数和调用父合约的方式是非常类似的，如库 L 有函数 f()，则使用 L.f()即可访问），库里面的内部函数被复制给使用它的合约。

同样按调用内部函数的调用方式，这意味着所有内部类型可以传进去，memory 类型则通过引用传递，而不是拷贝的方式。同样库里面的结构体和枚举也会被复制给使用它的合约。因此，如果一个库里只包含内部函数或结构体或枚举，则不需要部署库，因为库里面的所有内容都被复制给使用它的合约了。

下面的例子展示了如何使用库。

```
pragma solidity ^0.4.16;

library Set {
  // 定义了一个结构体，保存主调函数的数据（本身并未实际存储的数据）
  struct Data { mapping(uint => bool) flags; }

  // self 是一个存储类型的引用（而不是拷贝的值），这是库函数的特点
  // 参数名为 self 也是一个惯例，就像调用一个对象的方法一样
  function insert(Data storage self, uint value)
      public
      returns (bool)
```

```
    {
        if (self.flags[value])
            return false; // 已存在
        self.flags[value] = true;
        return true;
    }

    function remove(Data storage self, uint value)
        public
        returns (bool)
    {
        if (!self.flags[value])
            return false;
        self.flags[value] = false;
        return true;
    }

    function contains(Data storage self, uint value)
        public
        view
        returns (bool)
    {
        return self.flags[value];
    }
}

contract C {
    Set.Data knownValues;

    function register(uint value) public {
        // 库函数不需要实例化就可以调用，因为实例就是当前的合约
        require(Set.insert(knownValues, value));
    }
    // 在这个合约中，如果需要的话，可以直接访问 knownValues.flags
}
```

当然，我们也可以不按上面的方式来使用库函数，可以不定义结构体，可以不使用 storage 类型引用的参数，还可以在任何位置有多个 storage 引用类型

参数。

调用 Set.contains、Set.remove、Set.insert 会编译为以 DELEGATECALL 的方式调用外部合约和库。使用库时，需要注意的是一个真实的外部函数调用发生了。尽管 msg.sender、msg.value、this 还会保持它们在主调合约中的值（在 Homestead 之前，由于实际使用的是 CALLCODE，msg.sender、msg.value 会变化）。

下面的例子演示了在库中如何使用 memory 类型和内部函数来实现一个自定义类型，而不会用到外部函数调用。

```
pragma solidity ^0.4.16;

library BigInt {
    struct bigint {
        uint[] limbs;
    }

    function fromUint(uint x) internal pure returns (bigint r) {
        r.limbs = new uint[](1);
        r.limbs[0] = x;
    }

    function add(bigint _a, bigint _b) internal pure returns (bigint r) {
        r.limbs = new uint[](max(_a.limbs.length, _b.limbs.length));
        uint carry = 0;
        for (uint i = 0; i < r.limbs.length; ++i) {
            uint a = limb(_a, i);
            uint b = limb(_b, i);
            r.limbs[i] = a + b + carry;
            if (a + b < a || (a + b == uint(-1) && carry > 0))
                carry = 1;
            else
                carry = 0;
        }
        if (carry > 0) {
            // too bad, we have to add a limb
            uint[] memory newLimbs = new uint[](r.limbs.length + 1);
            for (i = 0; i < r.limbs.length; ++i)
```

```
                newLimbs[i] = r.limbs[i];
            newLimbs[i] = carry;
            r.limbs = newLimbs;
        }
    }

    function limb(bigint _a, uint _limb) internal pure returns (uint) {
        return _limb < _a.limbs.length ? _a.limbs[_limb] : 0;
    }

    function max(uint a, uint b) private pure returns (uint) {
        return a > b ? a : b;
    }
}

contract C {
    using BigInt for BigInt.bigint;

    function f() public pure {
        var x = BigInt.fromUint(7);
        var y = BigInt.fromUint(uint(-1));
        var z = x.add(y);
    }
}
```

在合约的源码中不能添加库地址，它是在编译时向编译器以参数形式提供的。这些地址必须由连接器（linker）填进最终的字节码中，使用命令行编译器来进行连接。如果地址没有以参数的方式正确给到编译器，编译后的字节码将仍会包含一个这样格式的占位符_Set___（其中 Set 是库的名称）。可以通过手动将所有的 40 个符号替换为库的十六进制数地址。

Using for 指令

指令 using A for B;用来把库函数（从库 A）关联到类型 B。这些函数将会把调用函数的实例作为第一个参数。语法与 Python 中的 self 变量一样。例如，库 A 有函数 add(B b1, B b2)，使用 Using A for B 指令后，如果有 B b1，就可以

使用 b1.add(b2)。

using A for * 表示库 A 中的函数可以关联到任意的类型上。

在这两种情形中，对于所有函数即使第一个参数的类型与调用函数的对象类型不匹配，也会被关联上。类型检查是在函数调用时及函数重载时执行的。

using A for B; 指令仅在当前的作用域有效，即目前仅仅支持当前合约的作用域，后续也非常有可能解除这个限制，允许作用到全局范围。如果能作用到全局范围，通过引入一些模块（module），数据类型将能通过库函数扩展功能，而不需要每个地方都写一遍类似的代码了。

下面我们使用 Using for 指令方式重写上一节 Set 的例子：

```
pragma solidity ^0.4.16;

// 库合约代码和上一节一样
library Set {
  struct Data { mapping(uint => bool) flags; }

  function insert(Data storage self, uint value)
      public
      returns (bool)
  {
      if (self.flags[value])
        return false; // already there
      self.flags[value] = true;
      return true;
  }

  function remove(Data storage self, uint value)
      public
      returns (bool)
  {
      if (!self.flags[value])
          return false; // not there
      self.flags[value] = false;
      return true;
  }

  function contains(Data storage self, uint value)
```

```
        public
        view
        returns (bool)
    {
        return self.flags[value];
    }
}

contract C {
    using Set for Set.Data; // 这是一个关键的变化
    Set.Data knownValues;

    function register(uint value) public {
        // 现在 Set.Data 都对应的成员方法
        // 效果和 Set.insert(knownValues, value)相同
        require(knownValues.insert(value));
    }
}
```

同样可以使用 Using for 的方式来对基本类型（elementary type）进行扩展：

```
pragma solidity ^0.4.16;

library Search {
    function indexOf(uint[] storage self, uint value)
        public
        view
        returns (uint)
    {
        for (uint i = 0; i < self.length; i++)
            if (self[i] == value) return i;
        return uint(-1);
    }
}

contract C {
    using Search for uint[];
    uint[] data;
```

```
    function append(uint value) public {
        data.push(value);
    }

    function replace(uint _old, uint _new) public {
        // 进行库调用
        uint index = data.indexOf(_old);
        if (index == uint(-1))
            data.push(_new);
        else
            data[index] = _new;
    }
}
```

需要注意的是所有库调用实际上是 EVM 函数调用。这意味着如果传的是 memory 类型或者值类型，那么进行拷贝时即使是 self 变量，解决方法也是使用存储（storage）类型的引用来避免拷贝内容的。

本章小结

合约是本书的重点之一。本章介绍了合约的概念、如何通过代码来创建合约、合约中的四种可见性，以及各种合约函数包括访问函数、视图函数、纯函数、回退函数，并且介绍了合约的继承和库的使用。

第 9 章

合约编译、部署、交互、调试

本章将介绍如何选用编译器对合约进行编译、不同环境下的合约部署、在合约部署之后如何调用合约，以及如何调试合约的方法。

Solidity 编译器

目前常用的 Solidity 编译器有三个：Remix、Solcjs、Solc。

Remix

Remix 是智能合约编程语言 Solidity 的集成开发环境，它集成了调试器和测试环境，是大家最常使用的编译器。Remix 主要提供的功能有：

1. 开发智能合约（集成了 Solidity 编辑器）。
2. 静态分析 Solidity 合约代码。
3. 部署合约、调试合约。

前面也介绍过，这是一个基于浏览器的在线 Solidity 集成开发环境（IDE），地址为 https://remix.ethereum.org/。

如果想离线使用，可以从 GitHub（https://github.com/ethereum/browser-solidity/tree/ gh-pages）上克隆一个。

当我们需要编写比较大的合约或是需要更多的编译选项时，建议使用命令行编译器 Solcjs 或 Solc。

Solcjs

Solcjs 是一个简化版的 Solc，它可以使用 npm 来安装。

```
npm install -g solc
```

安装完成之后，会有一个 Solcjs 命令。可以用以下命令验证安装是否成功。

```
> solcjs --help
Usage: solcjs [options] [input_file...]

选项:
  --version          显示版本号                                    [布尔]
  --optimize         Enable bytecode optimizer.                    [布尔]
  --bin              Binary of the contracts in hex.               [布尔]
  --abi              ABI of the contracts.                         [布尔]
  --standard-json    Turn on Standard JSON Input / Output mode.    [布尔]
  --output-dir, -o   Output directory for the contracts.           [字符串]
  --help             显示帮助信息                                  [布尔]
```

Solcjs 允许在 node.js 中以编程的方式来编译 Solidity。

Solc

Solc 是一个功能强大的 Solidity 命令行编译器，它提供了很多编译选项。

Mac 平台的安装方式如下：

```
brew install solidity
```

Ubuntu 平台的安装方式如下：

```
sudo add-apt-repository ppa:ethereum/ethereum
sudo add-apt-repository ppa:ethereum/ethereum-dev
sudo apt-get update
sudo apt-get install solc
```

其他平台的安装方式可以查看以下两个网址，https://github.com/ethereum/solidity/releases 及 http://solidity.readthedocs.io/en/develop/installing-solidity.html#binary-packages。

合约编译

合约的源代码被编译为 EVM 字节码和 ABI 接口说明后，才能够使用 Web3 部署。部署合约其实是一个交易。这个交易没有目标地址，交易的附加数据是编译出来的 EVM 字节码。当处理该交易时，EVM 会将输入的数据作为代码执行。这时一个合约就被创建了。

还是以第 1 章中的合约作为例子：

```
pragma solidity ^0.4.0;

contract SimpleStorage {

    uint storedData;

    function set(uint x) public {
        storedData = x;
    }

    function get() public constant returns (uint) {
        return storedData;
    }
}
```

使用 Solc 编译

生成 ABI 信息：

```
>solc --abi first.sol

======= first.sol:SimpleStorage =======
Contract JSON ABI
[{"constant":false,"inputs":[{"name":"x","type":"uint256"}],"name":"set","ou
tputs":[],"payable":false,"stateMutability":"nonpayable","type":"function"},
{"constant":true,"inputs":[],"name":"get","outputs":[{"name":"","type":"uint
256"}],"payable":false,"stateMutability":"view","type":"function"}]
```

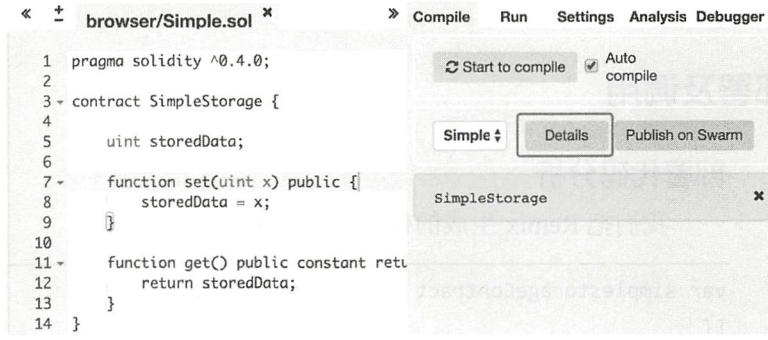
生成字节码：

```
>solc --bin first.sol

======= first.sol:SimpleStorage =======
Binary:
6060604052341561000f57600080fd5b60d38061001d6000396000f30060606
0405260043610604957600357c0100000000000000000000000000000000000
0000000000000000000000900463ffffffff16806360fe47b114604e5780636
d4ce63c14606e575b600080fd5b3415605857600080fd5b606c600480803590
6020019091905050606094565b005b341560785760080fd5b607e609e565b604
0518082815260200191505060405180910390f35b8060008190555050565b60
0080549050905600a165627a7a72305820ee31df166bd85697f9de5f2043e9a
31e571cb035933ae5c078dfb70ce9c5820b0029
```

使用 Remix 编译

把合约代码复制粘贴到 Remix 编译器中。Remix 编译器会直接为我们生成编译并用于部署的 web3.js 代码。这段代码可以通过以下方式查看到。

1. 点击 Compile 标签页的 Details，如下图所示。



2. 然后找到 WEB3DEPLOY。

ABI

```
▸ 0:
▸ 1:
```

WEB3DEPLOY

```
var simplestorageContract = web3.eth.contract([{"constant
var simplestorage = simplestorageContract.new(
   {
     from: web3.eth.accounts[0],
     data: '0x6080604052348015610010057600080fd5b5060df806
     gas: '4700000'
   }, function (e, contract){
    console.log(e, contract);
    if (typeof contract.address !== 'undefined') {
         console.log('Contract mined! address: ' + contra
    }
 })
```

合约部署及调用

部署代码分析

我们把 Remix 生成的代码拷贝出来分析一下：

```
var simplestorageContract = web3.eth.contract(
[{
    "constant": false,
    "inputs": [{
        "name": "x",
        "type": "uint256"
    }],
    "name": "set",
    "outputs": [],
    "payable": false,
    "stateMutability": "nonpayable",
```

```
    "type": "function"
}, {
    "constant": true,
    "inputs": [],
    "name": "get",
    "outputs": [{
        "name": "",
        "type": "uint256"
    }],
    "payable": false,
    "stateMutability": "view",
    "type": "function"
}]);
var simplestorage = simplestorageContract.new(
   {
     from: web3.eth.accounts[0],
     data:
'0x6080604052348015610010576000 80fd5b5060df8061001f6000396000f30060806040526
004361060495760003 57c0100000000000000000000000000000000000000000000000000000
000900463ffffffff16806360fe47b114604e5780636d4ce63c146078575b600080fd5b34801
5605957600080fd5b506076600480360381019080803590602001909291905050506 0a0565b0
05b348015608357600080fd5b50608a60aa565b60405180828152602001915050604051809 10
390f35b8060008190555050565b600080549050905600a165627a7a72305820efd5040d404c4
5d647fc1f87b2dd2e0266d605538798e4e9b99108ca6c004b960029',
     gas: '4700000'
   }, function (e, contract){
    console.log(e, contract);
    if (typeof contract.address !== 'undefined') {
         console.log('Contract mined! address: ' + contract.address + '
transactionHash: ' + contract.transactionHash);
    }
 })
```

第一行是用 web3.eth.contract 构造一个合约，它的参数是 ABI（ABI 信息也可由 Solc 生成）它包含了所有函数的说明；第二行是发起创建合约的交易，data 是这个交易的输入数据，即用于创建合约的 EVM 字节码（字节码也可由 Solc 生成）。另外，from 是发起交易的账户；gas 表示提供的 gas 量。

使用 geth

安装 geth

我们在第 2 章介绍过 geth。geth 的全称是 Go-ethereum，它是以太坊节点的 Go 语言实现版本。得益于 Go 语言的多平台特性，其支持在多个平台上使用，比如 Windows、Linux/Unix、Mac。geth 在不同平台的安装方法不一样，可参考官方链接：https://github.com/ ethereum/go-ethereum/wiki/Building-Ethereum。

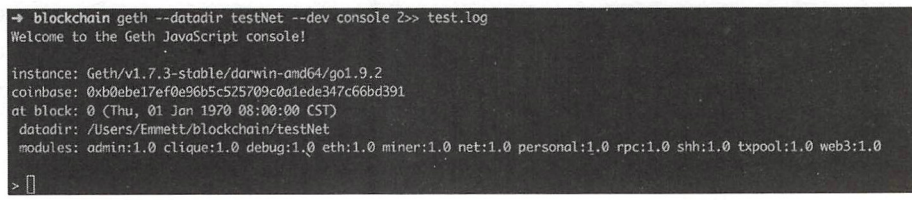
启动 geth

由于 geth 控制台同时也是 Web3.js 的运行环境（实现了所有 Web3 API），这段代码可以直接拷贝到 geth 控制台中运行并创建智能合约，下面介绍一下这个过程。

用下面的命令启动 geth。

```
geth --datadir testNet --dev console 2>> test.log
```

执行命令后，会进入 geth 控制台，这时光标停在一个向右的箭头处，如下图所示。



命令参数说明如下：

- --dev 启用开发者网络（模式），开发者网络会使用 PoA 共识，默认预分配一个开发者账户并且会自动开启挖矿。
- --datadir 后面的参数是区块数据及密钥存放目录。第一次输入命令后，它会在当前目录下新建一个 testNet 目录来存放数据。
- console 进入控制台。
- 2>> test.log 表示把控制台日志输出到 test.log 文件。

为了更好地理解，建议新开一个命令行终端，实时显示日志。

```
tail -f test.log
```

准备账户

部署智能合约需要一个外部账户。我们使用开发者模式启动 geth，它会自动分配一个开发者账户，在控制台中使用以下命令查看账户。

```
> eth.accounts
```

这个命令会返回所有的账号，如果没有账号，可以使用 personal. newAccount ("accountName")创建一个账号。同时确保账号中有余额，可以使用以下命令查看余额。

```
> eth.getBalance("accountAddr")
```

接着是解锁账号，只有在解锁账号后才能够进行支付，使用以下命令进行解锁。

```
> personal.unlockAccount("accountAddr", "pwd");
```

然后把解锁账号设置为默认账号。

```
> eth.defaultAccount = "accountAddr";
```

> 注意，上面命令中的 eth 及 personal 是 Web3.js 提供的 API 接口。

部署合约

把 Remix IDE 生产的代码拷贝到 geth 控制台，按下回车键运行就可以部署合约了。合约成功部署后会有提示，如下所示。

```
Contract mined! address: 0x9d77720ad98139952b33ff7de812957af811b09f
transactionHash:
0x04dc34342523f73e35c5b786c5119552b786660523a2cd5a9a43b414e6a174ca
```

这时在 geth 控制台，输入创建的合约实例 simplestorage，可以看到 simplestorage 的详细信息，如下所示。

```
> simplestorage
{
  abi: [{
      constant: false,
      inputs: [{...}],
      name: "set",
      outputs: [],
      payable: false,
      stateMutability: "nonpayable",
      type: "function"
  }, {
      constant: true,
      inputs: [],
      name: "get",
      outputs: [{...}],
      payable: false,
      stateMutability: "view",
      type: "function"
  }],
  address: "0x9d77720ad98139952b33ff7de812957af811b09f",
  transactionHash: "0x04dc34342523f73e35c5b786c5119552b786660523a2cd5a9a43b414e6a174ca",
  allEvents: function(),
  get: function(),
  set: function()
}
```

调用合约函数

在部署合约之后，可以直接使用实例 simplestorage 去调用函数，如下所示。

```
> simplestorage.set(3)
> simplestorage.get()
```

> 如果运行出错，请检查一下是否解锁或设置过默认账号。因为运行 set 需要消耗 gas，gas 会从默认账号中扣除。

使用 Remix + MetaMask

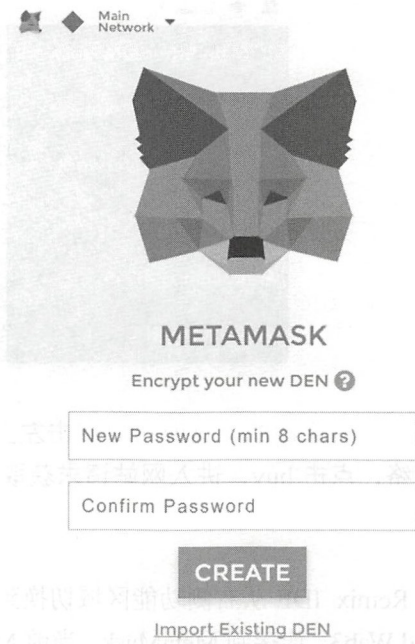
另外一个方法是使用 Remix + MetaMask 部署，我们来看看这个过程。

安装 MetaMask

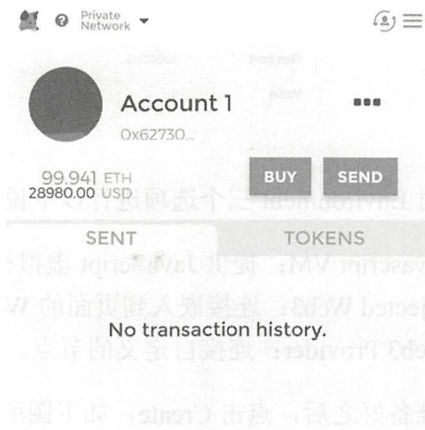
MetaMask 是一款浏览器插件形式的以太坊轻客户端，我们可以在开发过程中使用 MetaMask 和 DApp 进行交互。在官网 https://metamask. io/下载安装 MetaMask。安装完成后，浏览器工具条会显示一个小狐狸图标。

配置 MetaMask

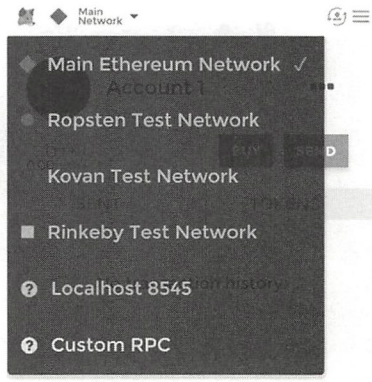
在接受隐私说明后，会出现以下页面。



然后输入想要的钱包密码（请妥善保管），点击 Create，进入 MetaMask 主界面，如下所示。



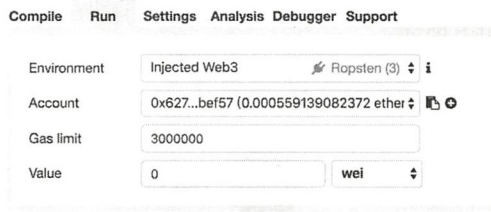
主界面的顶部，右上角账号小人图标是账号管理，可创建账号、导入账号、切换账号，左上角是网络管理，如下图所示。



> 新创建账号是没有余额的，这时可以点击左上角位置，切换到测试网络。如果是测试网络，点击 buy，进入网站请求获取一些以太币。

合约部署

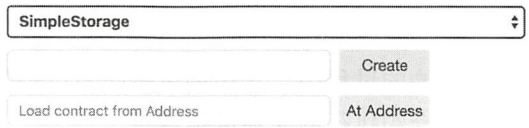
接下来把 Remix IDE 从右侧功能区域切换到 Run 的 tab 下，将 Environment 切换成 Injected Web3，连接到 MetaMask。当前 MetaMask 连接的是 Ropsten 网络（大家可以根据需要用上图的方法，选择不同的网络），如下图所示。



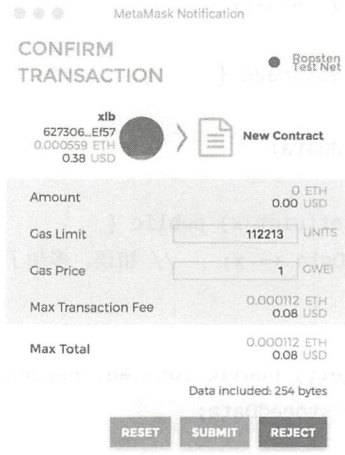
这里对 Environment 三个选项进行以下说明。

- Javascript VM：提供 JavaScript 虚拟机环境。
- Injected Web3：连接嵌入到页面的 Web3，如连接到 MetaMask。
- Web3 Provider：连接自定义的节点。

环境准备好之后，点击 Create，如下图所示。



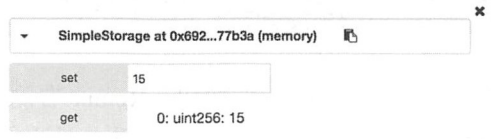
这时会弹出来一个交易确认框，因为在创建合约时，需要付费，这时会提示用户确认 gas 价格，如下图所示。



在提示框点击“SUBMIT”之后，创建合约交易，将由 Remix IDE 自动创建交易并发送到网络让矿工打包。矿工打包完成之后，IDE 会显示创建的合约信息，包含合约地址及合约函数，如下图所示。

调用合约函数

在 Remix IDE 中只需要点击函数名即可进行调用，如下图所示。



合约调试

在编写合约的过程中，合约调试是必不可少的一环，本节通过 Remix 来介绍一下如何对合约执行的交易进行调试。

为了模拟调试的过程，在前面的代码中故意加入了一段错误的逻辑，修改后的代码如下：

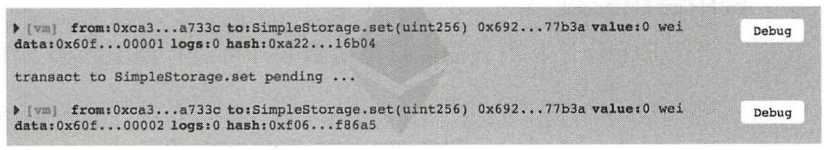
```
pragma solidity ^0.4.0;

contract SimpleStorage {

    uint storedData;

    function set(uint x) public {
        storedData += x;    // 错误，多加了一个加号
    }

    function get() public constant returns (uint) {
        return storedData;
    }
}
```

加入了错误的逻辑之后，第二次调用 set 函数，合约状态变量的值可能会出错（如果第一次不是用参数 0 去调用的话）。

注意，在部署合约的环境时应该选择：JavaScript VM。

开始调试

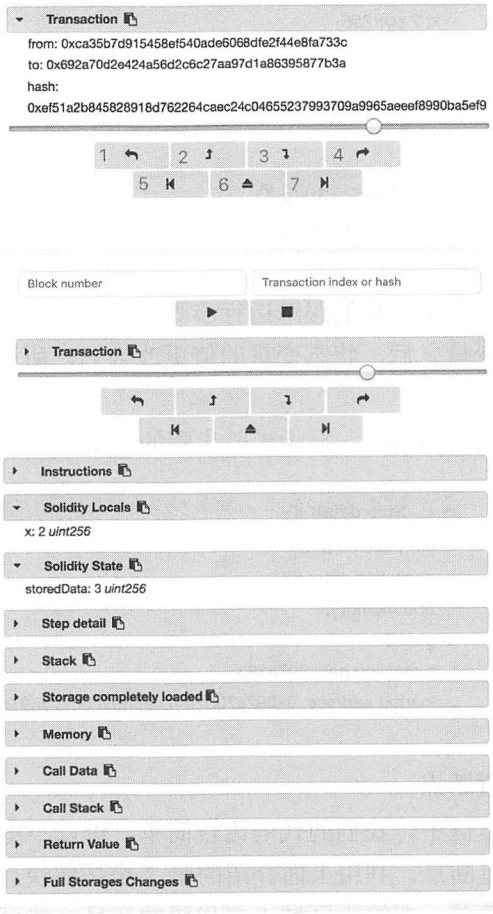
每次执行一个交易（不管是方式调用还是函数执行）时，日志都会输出一条记录，如下图所示。



点击上图中的“Debug”按钮，Remix 右侧的功能区域会切换到调试面板。

在调试过程中，有下面几项需要重点关注。

- Transactions：可以查看交易及交易的执行过程，并且提供了 7 个调试的按钮，如下图所示。



为了方便介绍，我为每个按钮编了号，每个按钮的含义如下。

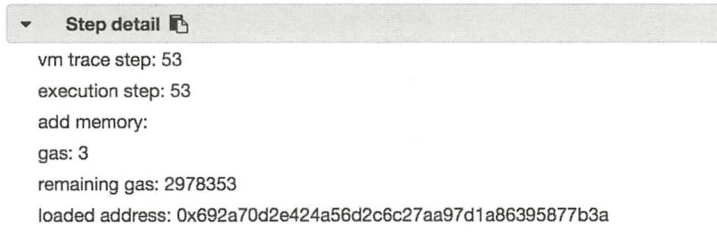
1. 后退一步（不进入函数内部）
2. 后退一步（进入函数内部）
3. 前进一步（进入函数内部）
4. 前进一步（不进入函数内部）
5. 跳到上一个断点
6. 跳出当前调用
7. 跳到下一个断点

- Solidity Locals：显示当前上下文的局部变量的值，如下图所示。

- Solidity State：显示当前执行合约的状态变量，如下图所示。

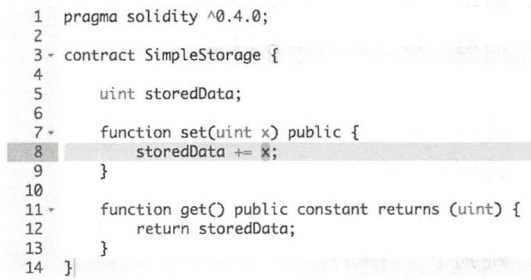
在本例中，我们跟踪运行步骤的时候，可以看到局部变量的值为 2，赋值给状态变量之后，状态变量的值更改为 3，所以可以判断运行当前语句的时候出错了。

- Step detail：显示当前步骤的 gas 详情等，如下图所示。



设置断点

在本例中，我们的代码比较简单，可以不用设置断点。如果代码比较多，可以设置断点。利用上面介绍的第 7 个按钮快速地调转到断点处，设置断点的方法很简单，在编辑区域点击代码的行号，就可以在当前行设置一个断点，如下图所示。



```
1  pragma solidity ^0.4.0;
2
3  contract SimpleStorage {
4
5      uint storedData;
6
7      function set(uint x) public {
8          storedData += x;
9      }
10
11     function get() public constant returns (uint) {
12         return storedData;
13     }
14 }
```

上图在第 8 行处设置了一个断点。成功设置断点后会在行号处加上一个色块来标示断点。取消断点的方式是再次点击断点处。

有一点需要注意一下，如果在声明变量的地方设置断点，那么这个断点可能会被触发两次。第一次是初始化为 0，第二次是赋实际的值。

本章小结

本章介绍了使用不同的工具来编译和部署合约，以及在不同的工具下如何调用合约函数，最后还重点介绍了如何使用 Remix 来调试合约。

第 10 章

应用程序二进制接口（ABI）

在以太坊（Ethereum）生态系统中，应用程序二进制接口（Application Binary Interface，ABI）是从区块链外部与合约进行交互，以及合约与合约之间进行交互的一种标准方式。简单来说就是以太坊调用合约时的接口说明，即定义操作函数签名、参数编码、返回结果编码等。当合约被编译后，那么它的 ABI 也就确定了。

简单理解 ABI

我们再次以第 1 章中的合约为例来看看其对应的 ABI。

```
pragma solidity ^0.4.0;

contract SimpleStorage {

    uint storedData;

    function set(uint x) public {
        storedData = x;
    }

    function get() public constant returns (uint) {
```

```
        return storedData;
    }
}
```

经过编译之后，它的字节码如下：

0x608060405234801561001057600080fd5b5060df8061001f6000396000f3006080604052600436106049576000357c0100000000000000000000000000000000000000000000000000000000900463ffffffff16806360fe47b114604e5780636d4ce63c146078575b600080fd5b348015605957600080fd5b506076600480360381019080803590602001909291905050506060a0565b005b34801560835760080fd5b50608a60aa565b60405180828152602001915050604051809103905f35b8060008190555050565b600080549050905600a165627a7a72305820efd5040d404c45d647fc1f87b2dd2e0266d605538798e4e9b99108ca6c004b960029

这个合约，我已经部署在 ropsten 网络上了，可以从以下链接看到输入的字节码信息。

https://ropsten.etherscan.io/tx/0x9e2134456becfd5072f0d1683a15e09683cf0d92d09dbfdded08162efb25ec68。

对于 EVM 而言，交易的输入数据是一个字节序列，合约的 ABI 就是指定了字节序列的编码模式。我们来看一个调用函数的例子就容易理解了，这个交易同样可以在下面这个链接查询到。

https://ropsten.etherscan.io/tx/0xf3342ff02420f830c22e48b6c8851b1c082b3cc5c3a9732167b063b603e4b42d

可以看到输入的字节码数据如下：

0x60fe47b1000000000000000000000000000000000000000000000000000000000000000001

那么 EVM 如何把 36 字节的数据解释为方法调用呢？答案就是依靠 ABI。这个输入数据可以分成两个子部分：

- 函数选择器（4 字节）0x60fe47b1
- 第一个参数（32 字节）0000000000000000000000000000000000000000000000000000000000000001

ABI 手册

关于 ABI 手册，这部分主要参考官方英文文档（https://solidity.readthedocs.

io/en/v0.4.24/abi-spec.html），由于 ABI 手册内容比较细碎，如果有介绍不到位的地方，欢迎大家向我反馈。

ABI 基本设计思想

使用 ABI 时必须要求在编译时知道类型，不支持动态语言那样的声明变量还会变类型的情况。也不支持动态合约接口，也就是不能在运行时推导出类型。

ABI 编码并不是可以自由描述的，而是需要一种特定的规则协议来进行解码，数据会根据其类型按照手册中说明的方法进行编码。

函数选择器

一个函数调用的前 4 个字节（高位在左的大端序）数据指定了要调用的函数签名。计算方式是使用函数签名的 keccak256（SHA-3）的哈希，取 4 个字节。

```
bytes4(keccak256("foo(uint32,bool)"))
```

函数签名使用基本原型的表达式，即函数名称加上由括号括起来的参数类型列表。如果有多个参数使用“,”隔开，则要去掉表达式中的所有空格。

注意：返回类型并不是函数签名的一部分，不过 ABI 里是包含了输入及输出的。

参数编码

由于前面的函数签名使用了 4 个字节，参数的数据将从第 5 个字节开始。参数的编码方式与返回值、事件的参数编码方式一致，后面一起介绍。

支持的类型

支持的基本类型，如下所示。

- uint<M>：无符号整型。M 为 8 的整数倍，并且在 0 到 256 之间，如 uint32、uint8、uint256。
- int<M>：整型。M 为 8 的整数倍，并且在 0 到 256 之间。
- address：ABI 层面等于 uint160。不过 address 有利于他人阅读和理解代码。
- uint, int：等于 uint256、int256。在计算函数选择器时使用 uint256、int256。
- bool：等于 uint8。但限制了值为 0 和 1，在计算函数选择器时使用的是 bool。
- fixed<M>x<N>：有符号固定位浮点数。M 为 8 的整数倍，并且在 0 到

256 之间, N 为 0 到 80 之间，其值 v 表示为 v / (10 ** N)。

- ufixed<M>x<N>：无符号的 fixed<M>x<N>。
- fixed, ufixed：等于 fixed128x19、ufixed128x19。在计算函数选择器时使用的是 fixed128x19 和 ufixed128x19。
- bytes<M>：M 个字节，0 < M <= 32。
- function：接在函数选择器（4 个字节）之后的 20 个字节的地址，被编码为 bytes24。

支持的固定长度的数组，如下所示。

- <type>[M]：M 个元素（类型为 type）的固定长度的数组，M > 0。

支持的不固定长度的数组，如下所示。

- bytes：字节序列。
- string：字符串。
- <type>[]：变长数组

需要存储的空间是不固定的。

支持不同类型合并为元组，如下所示。

- (T1,T2,…,Tn)：不同类型合并为元组，支持数组的元组、元组的元组，等等。元组也被用来编码结构体。

编码形式化说明

下面是正式编码的说明，它具有下面两个属性，如果参数是嵌套数组，这些属性尤其有用。

1. 访问一个参数属性需要的读取次数，取决于一个数组结构中最大的深度，比如 a_i[k][l][r]需要读取 4 次。在之前的 ABI 协议版本中，在最差情况下读取次数会随着总的动态参数的数量线性增长。

2. 一个变量的值或数组的元素的数据间不应该是隔开存储的（不会被插入其他的数据），并且是可以支持再定位的，比如仅使用相对的“地址”。

编码区分了动态类型（Dynamic Type）和静态类型（Static Type，也称为固定大小类型）。静态类型会就地直接编码，动态类型则会在当前数据块之后单独分配的位置被编码。

动态类型，如下所示。

- bytes：字节序列。
- string：字符串。
- T[]：任意类型的不定长数组。

- T[k]（k > 0）：任意类型的定长数组。
- (T1,…,Tk)：Ti 是动态类型的元组。

所有其他类型则为静态类型。

定义：对任意 ABI 值 X，我们根据 X 的实际类型递归定义 enc(X)。

- (T1,…,Tk) 对于 k >= 0 且任意类型 T1 ,…, Tk。

  enc(X) = head(X(1)) … head(X(k-1)) tail(X(0)) … tail(X(k-1))

这里，X(i)是元组 tuple 的第 i 个要素，并且当 Ti 为静态类型时，head 和 tail 被定义为 head(X(i)) = enc(X(i)) and tail(X(i)) = “”（空字符串）。

否则，被定义为 head(X(i)) = enc(len(head(X(0)) … head(X(k-1)) tail(X(0)) … tail(X(i-1))))tail(X(i)) = enc(X(i)), Ti 是动态类型。

注意，在动态类型的情况下，由于 head 部分的长度仅取决于类型而非值，所以 head(X(i))是定义明确的。它的值是从 enc(X) 的开头算起的，tail(X(i)) 的起始位置位于 enc(X)中的偏移量。

- T[k] 对于任意类型 T 和 k：enc(X) = enc((X[0],…,X[k-1]))。

它的编码与由相同类型的 k 个元素组成的元组是一样的。

- T[] 当 X 有 k 个元素（k 类型为 uint256）：enc(X) = enc(k) enc([X[1],…,X[k]])。

它就像是由静态大小为 k 的数组那样被编码的，并且由元素的个数作为前缀。

- 具有 k（类型为 uint256）长度的 bytes：enc(X) = enc(k) pad_right(X)，即字节数被编码为 uint256 紧跟着实际的 X 的字节码序列，再在前边（左边）补上可以使 len(enc(X))成为 32 的倍数的最少数量的 0 值字节数据。
- string：enc(X) = enc(enc_utf8(X))，X 被 utf-8 编码，并且在后续编码中将这个值解释为 bytes 类型。注意，在随后的编码中使用的长度是其 utf-8 编码的字符串的字节数，而不是其字符数。
- uint：enc(X) 是在 X 的大端序编码的前边（左边）补充若干 0 值字节以使其长度成为 32 的倍数。
- address：与 uint160 的情况相同。
- int：enc(X) 是在 X 的大端序 2 的补码编码的高位（左侧）添加若干字节数据以使其长度成为 32 的倍数；对于负数，添加值为 0xff（即 8 位全为 1）的字节数据；对于正数，添加 0 值（即 8 位全为 0）字节数据。
- bool：与 uint8 的情况相同，1 用来表示 true，0 表示 false。
- fixed<M>x<N>：enc(X) 就是 enc(X * 10**N)，其中 X * 10**N 可以理

解为 int256。

- fixed：与 fixed128x18 相同。
- ufixed<M>x<N>：enc(X)就是 enc(X * 10**N)，其中 X * 10**N 可以理解为 uint256。
- ufixed：与 ufixed128x18 的情况相同。
- bytes<M>：enc(X)就是 X 的字节序列加上为使长度称为 32 的倍数而添加的若干 0 值字节。

注意，对于任意的 X，len(enc(X))都是 32 的倍数。

函数选择器和参数编码

总的来说，对 f 函数的参数 a_1,…,a_n 按以下方式编码：

```
function_selector(f) enc((a_1,..., a_n))
```

f 函数对应的返回值 v_1,…,v_k 编码如下：

```
enc((v_1,..., v_k))
```

这里的[a_1,…, a_n]和[v_1,…, v_k]是定长数组，长度分别是 n 和 k。[a_1,…, a_n]可以是一个包含不同类型元素的数组。

下面是例子：

```
pragma solidity ^0.4.16;
contract Foo {
  function bar(fixed[2] xy) {}
  function baz(uint32 x, bool y) returns (bool r) { r = x > 32 || y; }
  function sam(bytes name, bool z, uint[] data) {}
}
```

以上这个例子如果要调用 baz(69, true)，一共要传 68 个字节，拆解如下：

- 0xcdcd77c0：使用函数选择器确定的函数 ID，bytes4(keccak256 ("baz(uint32,bool)"))。
- 0x0000000000000000000000000000000000000000000000000000000000000045。第一个参数 uint32 位的值是 69 并补位到 32 字节。
- 0x0000000000000000000000000000000000000000000000000000000000000001。第二个参数 boolean 类型值为 true。补位到 32 字节。

所以最终的串值为：

```
0xcdcd77c000000000000000000000000000000000000000000000000000000000000045000000000000000000000000000000000000000000000000000000000000001
```

返回结果是一个 bool 值，这里返回的是 false。所以输出的就是一个 bool。

```
0x0000000000000000000000000000000000000000000000000000000000000000
```

如果要调用 sam("dave",true,[1,2,3])，一共要传 292 个字节，拆解如下：

- 0xa5643bf2：函数 ID。bytes4(keccak256(sam(bytes,bool,uint256[]))。
- 0x0000000000000000000000000000000000000000000000000000000000000060：第一个参数（动态类型）的数据部分的位置，即从参数编码块开始位置算起的字节数。这里是 0x60。
- 0x0000000000000000000000000000000000000000000000000000000000000001：第二个参数 boolean 类型值为 true。
- 0x00000000000000000000000000000000000000000000000000000000000000a0：第三个参数（动态类型）的数据部分的位置，由 bytes 计算。
- 0x0000000000000000000000000000000000000000000000000000000000000004：第一个参数的数据部分，以字节数组的元素个数作为开始，在这里是 4。
- 0x6461766500000000000000000000000000000000000000000000000000000000：第一个参数的内容——“dave”的 UTF-8 编码（在这里等同于 ASCII 编码），并且在右侧（低位）用 0 值字节补充到 32 字节。
- 0x0000000000000000000000000000000000000000000000000000000000000003：第三个参数的数据部分，以数组的元素个数作为开始，这里是 3。
- 0x0000000000000000000000000000000000000000000000000000000000000001：第三个参数的第一个数组元素。
- 0x0000000000000000000000000000000000000000000000000000000000000002：第三个参数的第二个数组元素。
- 0x0000000000000000000000000000000000000000000000000000000000000003：第三个参数的第三个数组元素。

合起来就是：

```
0xa5643bf2000000000000000000000000000000000000000000000000000000000000006000000000000000000000000000000000000000000000000000000000000001000000000000000
```

```
0000000000000000000000000000000000000000000000a0000000000000000000000000
0000000000000000000000000000000000000000464617665000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000000000
0000000000000300000000000000000000000000000000000000000000000000000000000
0100000000000000000000000000000000000000000000000000000000000000200000000
0000000000000000000000000000000000000000000000000000000000000003
```

使用动态类型

如果要使参数为(0x123, [0x456, 0x789], "1234567890", "Hello, world!")，可调用函数 f(uint,uint32[],bytes10,bytes)。编码拆解如下。

bytes4(sha3("f(uint256,uint32[],bytes10,bytes)")) 计算 MethodID 值，得到 0x8be65246，然后对 4 个参数的头部进行编码。对于静态类型值 uint256 和 bytes10，直接编码传值。而对于动态内容类型值 uint32[]和 bytes，使用的字节数偏移量是它们的数据区域的起始位置，由编码值的开始位置算起（也就是说，不计算前 4 个用于表示函数签名的字节）。所以依次为：

- 0x0000000000000000000000000000000000000000000000000000000000000123，补位后的 32 字节 0x123。
- 0x0000000000000000000000000000000000000000000000000000000000000080，第二个参数由于是动态内容类型，所以这里存储偏移值，4×32 字节刚好是头部的大小。
- 0x3132333435363738393000000000000000000000000000000000000000000000，“1234567890”在右侧补 0 到 32 字节大小。
- 0x00000000000000000000000000000000000000000000000000000000000000e0，第四个参数的偏移 = 第一个动态参数的偏移值 + 第一个动态参数的长度= 4×32 + 3×32 动态长度的计算见下面。

尾部的第一个动态参数，[0x456, 0x789]编码拆解如下：

- 0x0000000000000000000000000000000000000000000000000000000000000002，整个数组的长度为 2。
- 0x0000000000000000000000000000000000000000000000000000000000000456，第一个元素。
- 0x0000000000000000000000000000000000000000000000000000000000000789，第二个元素。

最后来看一下第二个动态参数的编码，“Hello, world!”。

- 0x0000000000000000000000000000000000000000000000000000000000000000

00000d，元素的字节长度为 13。

- 0x48656c6c6f2c20776f726c6421000000000000000000000000000000000000000000，“Hello, world!”补位到 32 字节，里面是按 ASCII 编码的，可以查看对应的编码。

最终我们得到了下述的编码（函数签名的 4 个字节后的换行是为了显示清晰而加上的）：

```
0x8be65246
0000000000000000000000000000000000000000000000000000000000000123
0000000000000000000000000000000000000000000000000000000000000080
3132333435363738393000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000e0
0000000000000000000000000000000000000000000000000000000000000002
0000000000000000000000000000000000000000000000000000000000000456
0000000000000000000000000000000000000000000000000000000000000789
000000000000000000000000000000000000000000000000000000000000000d
48656c6c6f2c20776f726c6421000000000000000000000000000000000000000000
```

事件（Event）

事件是抽象出来的以太坊的日志项、事件监听协议。日志项包含合约的地址，最多可以达到 4 个主题和任意长度的二进制数据内容。事件也同样依赖 ABI 函数来解释，日志项被当成一个自定义数据结构。

事件有一个给定的事件名称，一系列的事件参数，参数分为两个类别：已索引的和未索引的。已索引的部分可以最多允许有三个，以及使用 Keccak 哈希算法计算的事件签名来组成日志主题。那些未索引的部分组成了事件的字节数组。

一个使用 ABI 描述的日志项，描述如下：

- address：合约的地址，由以太坊内部提供。
- topics[0]：keccak(EVENT_NAME+"("+EVENT_ARGS.map(canonical_type_of).join(",")+")")，其中的 canonical_type_of 是一个可以返回规范类型（Canonical form）的函数，如 uint indexed foo，返回的是 uint256。如果事件本身是匿名定义的，那么 Topic[0]将不会自动生成。
- Topics[n]：EVENT_INDEXED_ARGS[n-1]，其中的 EVENT_INDEXED_ARGS 表示已索引的事件参数。
- data：abi_serialise(EVENT_NON_INDEXED_ARGS)使用 ABI 协议序列

化的未索引的其他参数。abi_serialise()是 ABI 序列函数，用来返回一系列函数定义的类型值。

对于所有定长的 Solidity 类型，EVENT_INDEXED_ARGS 数组都会直接包含 32 字节的编码值。然而，对于动态长度的类型包含 string、bytes 和数组，EVENT_INDEXED_ARGS 会包含编码值的 Keccak 哈希而不是直接包含编码值。这样就允许应用程序更有效地查询动态长度类型的值（通过把编码值的哈希设定为主题），但也使应用程序不能对它们还没查询过的已索引的值进行解码。

对于动态长度的类型，应用程序开发者面临对预先设定的值（如果参数已被索引）的快速检索和对任意数据的清晰处理（需要参数不被索引）之间的权衡。开发者们可以通过定义两个参数（一个已索引、一个未索引）保存同一个值的方式来解决这种权衡，这样不仅能获得高效检索还能清晰地处理任意数据。

JSON 格式

合约接口的 JSON 格式是由一系列函数或事件的描述的数组来表示的。一个描述函数的 JSON 包含下述字段。

- type：可取值有 function、constructor、fallback（无名称的默认函数）。
- inputs：一系列对象，每个对象包含下述属性。
  - name：参数名称。
  - type：参数的规范类型。
  - components：供元组类型使用。
- outputs：一系列类似 inputs 的对象，无返回值时，可以省略。
- payable：true 表示函数可以接收 Ether，否则表示不能接收。

stateMutability：为下列值之一。pure（指定为不读取区块链状态），view（指定为不修改区块链状态）。

- constant：true 表示函数声明自己不会改变状态变量的值。

默认是 function 类型时，字段可以省略。构造函数和回退函数没有 name 或 outputs。回退函数也没有 inputs。

向不支持 payable 发送 Ether 将会引发异常，不应该这么做。

事件用 JSON 描述时和描述函数差不多。

- type：总是 event。
- name：事件的名称。
- inputs：一系列的对象，每个对象包含下述属性。
  - name：参数名称。

- type：参数的规范类型。
- components：供元组类型使用。
- indexed：true 代表这个字段是日志主题，false 代表是日志数据段。
- anonymous：true 代表事件是匿名声明的。

来看一个例子：

```
pragma solidity ^0.4.0;

contract Test {
  function Test() public { b = 0x1234567890123456789012345678901 2; }
  event Event(uint indexed a, bytes32 b);
  event Event2(uint indexed a, bytes32 b);
  function foo(uint a) public { Event(a, b); }
  bytes32 b;
}
```

其 ABI JSON 格式如下：

```
 [{
"type":"event",
"inputs":
[{"name":"a","type":"uint256","indexed":true},{"name":"b","type":"bytes32","
indexed":false}],
"name":"Event"
}, {
"type":"event",
"inputs":
[{"name":"a","type":"uint256","indexed":true},{"name":"b","type":"bytes32","
indexed":false}],
"name":"Event2"
      }, {
"type":"function",
"inputs": [{"name":"a","type":"uint256"}],
"name":"foo",
"outputs": []
}]
```

处理元组类型

尽管名称被有意地不作为 ABI 编码的一部分，但将它们包含进 JSON 来显示给最终用户是非常合理的。其结构会按下列方式进行嵌套。

看下面的例子：

```
pragma solidity ^0.4.19;
pragma experimental ABIEncoderV2;

contract Test {
  struct S { uint a; uint[] b; T[] c; }
  struct T { uint x; uint y; }
  function f(S s, T t, uint a) public { }
  function g() public returns (S s, T t, uint a) {}
}
```

其 ABI JSON 格式如下：

```
[
 {
   "name": "f",
   "type": "function",
   "inputs": [
     {
       "name": "s",
       "type": "tuple",
       "components": [
         {
           "name": "a",
           "type": "uint256"
         },
         {
           "name": "b",
           "type": "uint256[]"
         },
         {
           "name": "c",
           "type": "tuple[]",
           "components": [
```

```
            {
              "name": "x",
              "type": "uint256"
            },
            {
              "name": "y",
              "type": "uint256"
            }
          ]
        }
      ]
    },
    {
      "name": "t",
      "type": "tuple",
      "components": [
        {
          "name": "x",
          "type": "uint256"
        },
        {
          "name": "y",
          "type": "uint256"
        }
      ]
    },
    {
      "name": "a",
      "type": "uint256"
    }
  ],
  "outputs": []
}
]
```

非标准打包模式

Solidity 支持一种非标准打包模式：

- 函数选择器不进行编码。
- 长度低于 32 字节的类型，既不会进行补 0 操作，也不会进行符号扩展。同时，动态类型会直接进行编码，并且不包含长度信息。

例如，对 int1、bytes1、uint16、string 用数值 −1、0x42、0x2424、“Hello, world! ”进行编码将生成以下结果。

```
0xff42242448656c6c6f2c20776f726c6421
^^ int1(-1)
^^ bytes1(0x42)
^^^^ uint16(0x2424)
^^^^^^^^^^^^^^^^^^^^^^^^^^ string(“Hello, world!”) without a length field
```

更具体地说，每个静态类型的大小都尽可能多地按它们的数值范围使用字节数，而动态大小的类型像 string、bytes 或 uint[]，在编码时没有包含其长度信息。这意味着一旦有两个动态长度的元素，编码就会变得有歧义了。

本章小结

程序二进制接口 ABI 是从区块链外部与合约进行交互的，是合约与合约之间进行交互的一种标准方式。其规范涉及的细节信息非常多，对于大部分读者来说了解本章第一节的内容就好。

第 11 章

智能合约最佳实践

本章我们来看看如何写出一个更好的智能合约，主要从编码规范及安全指导原则两个方面介绍。

编码规范

每一门语言都有其相应的编码规范，Solidity 也一样，下面为官方推荐的规范，供大家参考。

命名规范

避免使用

小写的 l、大写的 I、大写的 O 应该避免在命名中单独出现，因为很容易产生混淆。

合约、库、事件、枚举及结构体命名

合约、库、事件及结构体命名应该使用单词首字母大写的方式。这个方式也称为“帕斯卡命名法”或“大驼峰式命名法”，比如：SimpleToken、SmartBank、CertificateHashRepository、Player。

函数、参数、变量及修饰器

函数、参数、变量及修饰器应该使用首字母小写，后面字母大写的方式。

这个方式也称为“小驼峰式命名法”，是一种混合大小写的方式。

- 函数名：getBalance、transfer、verifyOwner、addMember。
- 参数和变量：initialSupply、senderAddress、account、isPreSale。
- 修饰器：onlyAfter、onlyOwner。

代码格式

缩进

使用空格（space）而不是 Tab，缩进应该是 4 个空格。

空行

合约之间应该有空行，例如：

```
contract A {
    ...
}


contract B {
    ...
}


contract C {
    ...
}
```

错误示例如下：

```
contract A {
    ...
}
contract B {
    ...
}
```

```
contract C {
    ...
}
```

函数之间应该有空行，例如：

```
contract A {
    function spam() public {
        ...
    }

    function ham() public {
        ...
    }
}
```

没有实现的话，空行可以省去，例如：

```
contract A {
    function spam() public;
    function ham() public;
}
```

错误示例如下：

```
contract A {
    function spam() public {
        ...
    }
    function ham() public {
        ...
    }
}
```

左括号应该跟定义在一行

定义包括合约定义、函数定义、库定义、结构体定义，等等。例如：

```
contract Coin {
    struct Bank {
        address owner;
```

```
        uint balance;
    }
}
```

错误示例如下：

```
contract Coin
{
    struct Bank {
        address owner;
        uint balance;
    }
}
```

左括号应该跟条件控制在一行。在使用 if、else、while、for 时，推荐的写法如下：

```
if (...) {
    ...
}

for (...) {
    ...
}
```

错误示例如下：

```
if (...)
{
    ...
}

while(...){
}

for (...) {
    ...;}
```

如果控制语句内只有一行，括号可省略，如下所示：

```
if (x < 10)
    x += 1;
```

但像下面这个语句有多行，括号就不能省略，如下所示：

```
if (x < 10)
    someArray.push(Coin({
        name: 'spam',
        value: 42
    }));
```

表达式内的空格

对于一个单行的表达式，在小括号、中括号、大括号里应该避免不必要的空格，例如：

```
spam(ham[1], Coin({name: "ham"}));
```

错误示例如下：

```
spam( ham[ 1 ], Coin( { name: "ham" } ) );
```

有一种例外是，结尾的括号跟在结束的分号后面，应该加一个空格。例如下面的方式也是推荐的：

```
function singleLine() public { spam(); }
```

分号“;”前不应该有空格，例如：

```
function spam(uint i, Coin coin) public;
```

错误示例如下：

```
function spam(uint i , Coin coin) public ;
```

不要为对齐添加不必要的空格，例如：

```
x = 1;
y = 2;
long_variable = 3;
```


错误示例如下：

```
x             = 1;
y             = 2;
long_variable = 3;
```

回退函数不应该有空格，例如：

```
function() public {
    ...
}
```

错误示例如下：

```
function () public {
    ...
}
```

控制每一行长度

每行不应该太长，最好在 79（或 99）个字符以内，函数的参数应该是单独成行的，并且只有一个缩进，例如：

```
thisFunctionCallIsReallyLong(
    longArgument1,
    longArgument2,
    longArgument3
);
```

错误示例如下：

```
thisFunctionCallIsReallyLong(longArgument1,
                             longArgument2,
                             longArgument3
);

thisFunctionCallIsReallyLong(longArgument1,
    longArgument2,
    longArgument3
);
```

```
thisFunctionCallIsReallyLong(
    longArgument1, longArgument2,
    longArgument3
);

thisFunctionCallIsReallyLong(
longArgument1,
longArgument2,
longArgument3
);

thisFunctionCallIsReallyLong(
    longArgument1,
    longArgument2,
    longArgument3);
```

对应的赋值语句应该是这样写：

```
thisIsALongNestedMapping[being][set][to_some_value] = someFunction(
    argument1,
    argument2,
    argument3,
    argument4
);
```

错误示例如下：

```
thisIsALongNestedMapping[being][set][to_some_value] = someFunction(argument1,
                                                                   argument2,
                                                                   argument3,
                                                                  argument4);
```

事件定义也应该遵循同样的原则，例如：

```
event LongAndLotsOfArgs(
    adress sender,
    adress recipient,
    uint256 publicKey,
    uint256 amount,
    bytes32[] options
```


```
);

LongAndLotsOfArgs(
    sender,
    recipient,
    publicKey,
    amount,
    options
);
```

错误示例如下：

```
event LongAndLotsOfArgs(adress sender,
                        adress recipient,
                        uint256 publicKey,
                        uint256 amount,
                        bytes32[] options);

LongAndLotsOfArgs(sender,
                  recipient,
                  publicKey,
                  amount,
                  options);
```

文件编码格式

推荐使用 UTF-8 及 ASCII 编码。

引入文件应该在最上方

建议使用的格式如下：

```
import "owned";


contract A {
    ...
}
```

```
contract B is owned {
    ...
}
```

错误示例如下：

```
contract A {
    ...
}


import "owned";


contract B is owned {
    ...
}
```

函数编写规范

函数的顺序

在编写函数的时候，应该让大家容易找到构造函数、回退函数。官方推荐的函数顺序如下。

1. 构造函数。
2. 回退函数。
3. 外部函数。
4. 公有函数。
5. 内部函数。
6. 私有函数。

当有同一类函数时，constant 函数放在后面，例如：

```
contract A {
    // 构造函数
    function A() public {
        ...
    }
```


```
    // 回退函数
    function() public {
        ...
    }

    // 外部函数
    // ...

    // 带有 constant 外部函数
    // ...

    // 公有函数
    // ...

    // 内部函数
    // ...

    // 私有函数
    // ...
}
```

而不是下面的函数顺序，错误示例如下：

```
contract A {


    // 外部函数
    // ...

    // 公有函数
    // ...

    // 内部函数
    // ...

    function A() public {
        ...
    }
```


```
    function() public {
        ...
    }

    // 私有函数
    // ...
}
```

明确函数的可见性

所有的函数（包括构造函数）应该在定义的时候明确函数的可见性，例如：

```
function explicitlyPublic(uint val) public {
    doSomething();
}
```

错误示例如下：

```
function implicitlyPublic(uint val) {
    doSomething();
}
```

可见性应该在修饰符前面

函数的可见性应该写在自定义的函数修饰符前面，例如：

```
function kill() public onlyowner {
    selfdestruct(owner);
}
```

错误示例如下：

```
function kill() onlyowner public {
    selfdestruct(owner);
}
```

区分函数和事件

为了防止函数和事件（Event）产生混淆，声明一个事件使用大写字母并加入前缀（可使用 Log）。而函数始终以小写字母开头，但构造函数除外。

```
// 不建议
event Transfer() {}
function transfer() {}

// 建议
event LogTransfer() {}
function transfer() external {}
```

常量

常量应该使用全大写字母及下画线分割大词的方式，如：MAX_BLOCKS、TOKEN_NAME、CONTRACT_VERSION。

安全性考虑

区块链是一把双刃剑，弊端在于它的不可篡改性。不像我们现有的互联网应用，可以轻松升级，现在的互联网思维要求我们不断试错，快速迭代。

在区块链上开发智能合约需要一个全新的思维，因为犯错的代价很大（比如造成财产的直接损失）。不能轻松升级会给开发带来更大的挑战。因此我们需要有新的开发理念。以下为 Consensys 总结的几点原则：

要对错误有所准备。任何合约都可能存在错误，需要我们在代码中预先处理出现的 bug 和漏洞。

1. 当智能合约出现错误时，智能合约失效。
2. 设置资金转移额度及速率限制。
3. 提供升级的方式。

谨慎发布智能合约。

1. 对智能合约进行彻底测试，并且在出现新的攻击方法后及时测试。
2. 先发布到测试网络。
3. 分阶段发布，每个阶段都提供足够多的测试。
4. 提供 bug 赏金计划。

保持智能合约的简洁。

1. 确保合约和函数模块化。
2. 使用已经被广泛使用的合约或工具。例如，不要自己写一个随机数生成器。
3. 有时清晰比性能更重要。
4. 只在系统中去中心化部分使用区块链。

保持更新。

1．在任何新的漏洞被发现时检查智能合约。

2．尽可能将库或者工具更新到最新。

3．使用最新的安全技术。

清楚区块链的特性。

1．外部合约的调用要格外小心，可能执行到一段恶意代码。

2．public 函数是公开的，意味着可以被恶意调用。

3．private 数据其实也是可见的。

4．清楚 gas 的花费和区块的 gas limit。

一些安全陷阱

私有信息和随机数

在智能合约中你所用的一切都是公开可见的，即便是局部变量和被标记成 private 的状态变量也是如此。

如果不想让矿工作弊的话，在智能合约中使用随机数会很棘手。在智能合约中使用随机数很难保证节点不作弊，这是因为智能合约中的随机数一般要依赖计算节点的本地时间得到，而本地时间是可以被恶意节点伪造的，因此这种方法并不安全。通行的做法是采用链外的第三方服务，比如用 Oraclize 来获取随机数。

重入问题

任何从合约 A 到合约 B 的交互，以及任何从合约 A 到合约 B 的以太币的转移，都会将控制权交给合约 B。这使得合约 B 能够在交互结束前回调 A 中的代码。举个例子，下面的代码中就出现了这样的 bug。

```
pragma solidity ^0.4.0;

// 不应该使用这个合约
contract Fund {
    // Mapping of ether shares of the contract
    mapping(address => uint) shares;
    // Withdraw your share
    function withdraw() public {
```

```
        if (msg.sender.send(shares[msg.sender]))
            shares[msg.sender] = 0;
    }
}
```

虽然这里的问题不是很严重，但仍然暴露了一个缺陷：在以太币的传输过程中可以包含代码执行，所以接收者可以一个回调进入 withdraw 的合约。这就会使其多次得到退款，从而将合约中的全部以太币取走。下面这个合约示例尤其危险，它将允许一个攻击者多次得到退款，因为它使用的 call 没有 gas 限制。

```
pragma solidity ^0.4.0;

//不应该使用这个合约
contract Fund {
    // Mapping of ether shares of the contract
    mapping(address => uint) shares;
    // Withdraw your share
    function withdraw() public {
        if (msg.sender.call.value(shares[msg.sender])())
            shares[msg.sender] = 0;
    }
}
```

为了避免重入，可以使用下面撰写的“检查—生效—交互”（Check-Effect-Interaction）模式。

```
pragma solidity ^0.4.11;

contract Fund {
    // Mapping of ether shares of the contract
    mapping(address => uint) shares;
    // Withdraw your share
    function withdraw() public {
        var share = shares[msg.sender];
        shares[msg.sender] = 0;
        msg.sender.transfer(share);
    }
}
```

重入问题不仅仅是发送以太币时才有，对每一个合约的调用都应该考虑这个问题。

gas 限制和循环

在使用没有固定迭代次数的循环时要非常小心。例如依赖变量值的循环，由于 gas 是有限制的，所示交易只能消耗一定数量的 gas。如果依赖变量，那么很可能使循环中的数次迭代操作所消耗的 gas 超出限制，从而导致整个合约在运行的某个时刻突然停止。

编写合约的安全建议

尽早且明确地暴露问题

一个简单且强大的最佳实践是——尽早且明确地暴露问题。我们看一个有问题的函数实现：

```
// 警告：不要使用这个合约代码
contract BadFailEarly {
  uint constant DEFAULT_SALARY = 50000;
  mapping(string => uint) nameToSalary;
  function getSalary(string name) constant returns (uint) {
    if (bytes(name).length != 0 && nameToSalary[name] != 0) {
      return nameToSalary[name];
    } else {
      return DEFAULT_SALARY;
    }
  }
}
```

上面例子中的函数 getSalary 为避免合约潜在的问题，在返回结果前检查了参数。那么这个例子有什么问题呢？问题在于，如果条件不满足，那么将返回默认值。这将掩盖参数的问题。因为其仍然可以按正常业务逻辑返回值。这虽然是一个比较极端的例子，但却非常常见。原因是大家在进行程序设计时，担心程序兼容性不够，所以会设置一些兜底方案。但真相是，越快失败，越容易发现问题。如果我们只想掩盖错误，那么错误将扩散到代码的其他地方，从而发生难以跟踪的不一致问题。下面是一段调整后的代码：

```
contract GoodFailEarly {
  mapping(string => uint) nameToSalary;

  function getSalary(string name) constant returns (uint) {
    require (bytes(name).length != 0)
    require (nameToSalary[name] != 0)
    return nameToSalary[name];
  }
}
```

还有一个方法是使用 Solidity 提供的修改器的特性将条件预检查分开，这样可以带来重用的好处。

函数代码的顺序：条件、行为、交互

另一个较好的实践是在实现函数时，按以下步骤写代码。

- 首先，检查所有前置的条件。
- 然后，对合约的状态进行修改。
- 最后，与其他合约进行交互。

条件、行为、交互，坚持使用这样的函数结构，将会使我们避免大部分的问题。下面来看一个使用了这个模式的例子：

```
function auctionEnd() {
  // 前置条件
  assert (now > auctionStart + biddingTime);
  assert (!ended)


  // 状态修改
  ended = true;
  AuctionEnded(highestBidder, highestBid);

  // 交互
  assert (beneficiary.send(highestBid))

}
```

合约要符合尽可能早地暴露问题的原则，因为条件在一开始就进行了检查。它将存在潜在交互风险的与其他合约的交互放到了最后。

注意平台的限制

EVM 有非常多的关于合约的隐形限制。这些是平台级的安全考虑，如果不知道的话，会威胁合约安全。下面我们来看一个看起来正常实际上却有问题的雇员津贴管理的例子：

```
contract BadArrayUse {

  address[] employees;

  function payBonus() {
    for (var i = 0; i < employees.length; i++) {
      address employee = employees[i];
      uint bonus = calculateBonus(employee);
      employee.send(bonus);
    }
  }

  function calculateBonus(address employee) returns (uint) {
    //
  }
}
```

上面的代码实现得非常简单、直接，表面上看没有什么问题。但由于 EVM 平台的一些独特性，其背后却潜藏着以下三个问题：

第一个问题是 i 的类型将会是 uint8。因为如果要存 0 却不指定类型，那么 EVM 将自动选择一个占用空间最小、最恰当的类型，在这个例子中选择的是 uint8。所以如果这个数组 employees 大小超过 255 个元素，那么这个循环将永远不会结束，最终导致 gas 耗尽。在定义变量时，尽可能不要使用 var，明确变量的类型。修正后的例子如下所示。

```
// 仍然是不安全的代码，请不要使用
contract BadArrayUse {

  address[] employees;
```

```
  function payBonus() {
    for (uint i = 0; i < employees.length; i++) {
      address employee = employees[i];
      uint bonus = calculateBonus(employee);
      employee.send(bonus);
    }
  }

  function calculateBonus(address employee) returns (uint) {
    // some expensive computation ...
  }
}
```

第二个问题是要考虑 gas 的限制。例如 calculateBonus 在计算津贴时需要进行复杂运算，比如跨多个项目计算利润。这将消耗非常多的 gas，很容易达到交易和区块的 gas 限制。如果一个交易达到了 gas 的限制，那么所有状态的改变将会撤销，但消耗的 gas 不会退回。当使用循环的时候，要特别注意变量对 gas 消耗的影响。现在优化一下上述代码，将津贴计算与循环分开。但需要注意的是，分开后仍然有数组变大后带来的 gas 消耗增长的问题。

```
// 仍然是不安全的代码，请不要使用
contract BadArrayUse {

  address[] employees;
  mapping(address => uint) bonuses;

  function payBonus() {
    for (uint i = 0; i < employees.length; i++) {
      address employee = employees[i];
      uint bonus = bonuses[employee];
      employee.send(bonus);
    }
  }

  function calculateBonus(address employee) returns (uint) {
    uint bonus = 0;
    // some expensive computation modifying the bonus...
```

```
    bonuses[employee] = bonus;
  }
}
```

最后一个问题是如果合约调用失败，将无法支付以太币。在支付时我们应该优先使用 pull（指用户自己去请求获取）方式而不是 push（指在合约内把币推送给用户）方式。下面我们进一步讲解该内容。

支付时优先使用 pull 而不是 push

每次以太币转移时都需要考虑对应账户可能是一个合约账户的情况，而接收合约会实现一个默认的回退函数。在收到以太币时，这个函数会被执行并可能抛出错误。因此，要考虑在 send 执行中可能出现的错误。一个解决方案是，在支付时使用 pull 而不是 push（不过 pull 在体验上比 push 差一些）。我们来看一个关于竞标函数的例子。

```
// 警告：不要使用这个合约代码
contract BadPushPayments {
  address highestBidder;
  uint highestBid;

  function bid() {
    if (msg.value < highestBid) throw;
    if (highestBidder != 0) {
      // return bid to previous winner
      if (!highestBidder.send(highestBid)) {
        throw;
      }
    }
    highestBidder = msg.sender;
    highestBid = msg.value;
  }
}
```

上述合约调用了 send 函数，检查了返回值，看起来是对的。但实际上却有隐藏的安全问题，为什么呢？因为 send 函数可能会触发另外一个合约的回退函数代码执行。

假如某个竞标的地址会在每次有人转账给它时 throw，那么当其他人尝试追

加价格竞标时会发生什么呢？send 调用将会失败，从而错误上抛，让 bid 函数产生一个异常。一个函数调用如果以错误结束，那么会让状态不发生变更（所有的变化都将回滚）。这意味着，没有人能继续竞标，合约因此失效。

最简单的解决方案是，将支付分离到另一个函数中，让用户请求（pull）金额，而不是依赖合约逻辑。

```
contract GoodPullPayments {
  address highestBidder;
  uint highestBid;
  mapping(address => uint) refunds;

  function bid() external {
    if (msg.value < highestBid) throw;

    if (highestBidder != 0) {
      refunds[highestBidder] += highestBid;
    }

    highestBidder = msg.sender;
    highestBid = msg.value;
  }

  function withdrawBid() external {
    uint refund = refunds[msg.sender];
    refunds[msg.sender] = 0;
    if (!msg.sender.send(refund)) {
      refunds[msg.sender] = refund;
    }
  }
}
```

上面的合约使用了一个 mapping 来存储每个待退款的竞标者的信息，提供了一个 withdraw 用于退款。如果在 send 调用时抛出异常，那么仅仅只是那个有问题的竞标者受到影响。这是一个非常简单的模式，却解决了很多问题（比如，可重入问题）。所以，记住一点，当发送 Ether 时，要使用 pull 而不是 push。

尽量避免外部调用

每一个外部调用都可能会有潜在的安全威胁，调用不受信任的外部合约可能会引发意外的风险和错误。外部调用可能在其合约和它所依赖的其他合约内执行恶意代码。在外部调用前应该明白所要执行的代码。

标记不信任的合约

当函数调用外部合约时，应尽量在变量、方法、合约接口命名时表明他们可能是不安全的。以下是一个例子：

```
// 不建议
Bank.withdraw(100);

// 不建议
function makeWithdrawal(uint amount) {
    Bank.withdraw(amount);
}

// 建议使用这种方式
UntrustedBank.withdraw(100);
TrustedBank.withdraw(100);

function makeUntrustedWithdrawal(uint amount) {
    UntrustedBank.withdraw(amount);
}
```

处理外部调用错误

Solidity 提供了一系列在 raw address 上执行操作的底层方法，比如：address.call()、address.callcode()、address.delegatecall()和 address.send。这些底层方法不会抛出异常，只是会在遇到错误时返回 false。另一方面，contract calls（比如，ExternalContract.doSomething()）会自动传递异常。doSomething()抛出异常，那么 ExternalContract.doSomething() 同样会进行 throw。

如果你选择使用底层方法，那么一定要检查返回值来对可能的错误进行处理。

send()、transfer()还是 call.value()()

当发送 Ether 时，需要仔细权衡 someAddress.send()、someAddress.transfer()

和 someAddress.call.value()()之间的差别。

someAddress.send()和 someAddress.transfer() 能保证可重入安全。尽管这些智能合约方法可以被触发执行，但提供给智能合约的 2300 gas 仅仅够发送一个事件。

x.transfer(y)和 if require(x.send(y))是等价的。send 是 transfer 的底层实现，建议尽可能直接使用 transfer，发送失败时自动回退。

someAddress.call.value(y)() 将发送指定数量的 Ether 并触发对应代码的执行。执行的代码将可用所有的 gas，通过这种方式发起的交易可能会有可重入性漏洞。

可重入性漏洞是当一个合约调用另一个合约的时候，当前执行进程就会停下来等待调用结束，这就导致了一个可以被利用的中间状态。利用合约存在的中间状态，当一个合约还没有调用完成时发起另一个调用交易，即可完成攻击。

下面是一个使用 call()引起的可重入漏洞：

```
contract Bank {
    function withdraw(){
        uint amount = balances[msg.sender] ;

        //  会触发回退函数
        if(msg.sender.call.value(amount)() == false){
            throw ;
        }
        balances[msg.sender] = 0 ;
    }
}

contract BadUser {
    function money(address addr){
        Bank(addr).withdraw();
    }

    function () payable {
        Bank(addr).withdraw() ;
    }
}
```

在上面的合约中，用户首先调用 money(), 执行的跨合约调用会回调到

BadUser 的回退函数。在 BadUser 的回退函数中会再次执行 withdraw()，此时的 msg.sender 是 Bank 合约本身，因此会把 Bank 合约的所有余额发送到 BadUser 上。这就是著名的 THeDAO 事件。

回退函数保持简单

回退函数在合约没有匹配的函数、执行消息发送没有携带参数或收到以太币时，将会被调用。尤其是当调用.send()或.transfer()时，通常只有 2300 gas 用于 fallback 函数的执行（如果希望能够监听.send()或.transfer()接收到 Ether，则可以在回退函数中使用 event）。所以我们应该保持回退函数足够简单，以避免 gas 不够用，下面是一个例子：

```
// 不建议
function() payable { balances[msg.sender] += msg.value; }

// 建议
function deposit() payable external { balances[msg.sender] += msg.value; }

function() payable {
 LogDepositReceived(msg.sender); }
```

限制可存入的资金

另一个保护智能合约远离攻击的方式是“限制”。攻击者最有可能针对管理着数百万美元的高额合同。并不是所有的合约都有这样大的资金量，尤其是在初期时。在这种情形下，限制合约可以接收的资金量就非常重要了。最简单的方式是设置一个余额上限。

```
contract LimitFunds {

  uint LIMIT = 5000;

  function deposit() payable {
   require (this.balance <= LIMIT);
    ...
  }
}
```

标明函数和状态变量的可见性

标明函数和状态变量的可见性。函数可以声明为 external、public、internal 或 private。分清楚它们之间的差异，如果 external 够用就不要使用 public。对于状态变量 external，标明可见性将更容易避免关于谁可以调用该函数或访问变量的错误假设。

```
// 不建议
uint x; // 默认是 private ，但应该显式声明
function buy() { // 默认是 public
    // public code
}

// 建议使用
uint private y;
function buy() external {
    // 只能外部调用
}

function utility() public {
    // 可外部调用 + 内部调用

}

function internalAction() internal {
    // 内部调用
}
```

本章小结

上述这些实践主要参考了以下文章。

- https://github.com/ ConsenSys/smart-contract-best-practices
- https://github.com/ethereum/wiki/wiki/ Safety#favor-pull-over-push-for-external-calls
- https://blog.zeppelin.solutions/onward-with-ethereum-smart-contract-security - 97a827e47702

- http://solidity. readthedocs.io/en/develop/style-guide.html

相信这些实践可以帮助我们写出更安全的合约。

同时推荐大家看看 GitHub（https://github.com/OpenZeppelin/zeppelin-solidityOpenZeppelin）上的一些库，来帮助我们写出更安全的智能合约。

第 12 章

合约案例

本章讲解了一些常用的合约案例代码。这些案例有代币合约 ERC20、众筹合约，以及 ERC721 合约的接口分析。

代币

代币（Token）现在也有人称为“通证”，利用以太坊的智能合约可以轻松编写、发行一个属于自己的代币，代币可以代表任何可以交易的东西，例如：积分、财产，等等。因此不论是出于商业目的，还是兴趣爱好，很多人都想创建一个属于自己的代币。我们先贴一张图看看创建的代币是什么样子的。

接下来，我们就来详细讲一讲如何创建一个代币。

ERC20 Token

ERC20 是以太坊定义的一个代币标准，要求我们在实现代币的时候必须要遵守的协议，如指定代币名称、总量、实现代币交易函数等，只有支持了协议才能被以太坊钱包支持。其接口如下：

```
contract ERC20Interface {

    string public constant name = "Token Name";
    string public constant symbol = "SYM";
    uint8 public constant decimals = 1;
    function totalSupply() public constant returns (uint);
    function balanceOf(address tokenOwner) public constant returns (uint
balance);
    function allowance(address tokenOwner, address spender) public constant
returns (uint remaining);
    function transfer(address to, uint tokens) public returns (bool success);
    function approve(address spender, uint tokens) public returns (bool success);
    function transferFrom(address from, address to, uint tokens) public returns
(bool success);

    event Transfer(address indexed from, address indexed to, uint tokens);
    event Approval(address indexed tokenOwner, address indexed spender, uint
tokens);
}
```

简单说明一下。

- name：代币名称。
- symbol：代币符号。
- decimals：代币小数点位数，代币的最小单位，1 表示我们可以拥有.1 单位个代币。
- totalSupply()：发行代币总量。
- balanceOf()：查看对应账号的代币余额。
- transfer()：实现代币交易，用于给用户发送代币（从我们的账户里）。
- transferFrom()：实现代币用户之间的交易。
- allowance()：控制代币的交易，例如可交易账号及资产。
- approve()：允许用户可花费的代币数。

编写代币合约代码

代币合约代码，如下所示：

```
pragma solidity ^0.4.16;

interface tokenRecipient { function receiveApproval(address _from, uint256
_value, address _token, bytes _extraData) public; }

contract TokenERC20 {
    string public name;
    string public symbol;
    uint8 public decimals = 18;  // decimals 为可以有的小数点个数，最小的代币单位
                                 // 18 是建议的默认值
    uint256 public totalSupply;

    // 用 mapping 保存每个地址对应的余额
    mapping (address => uint256) public balanceOf;
    // 存储对账号的控制
    mapping (address => mapping (address => uint256)) public allowance;

    // 事件，用来通知客户端交易发生
    event Transfer(address indexed from, address indexed to, uint256 value);

    // 事件，用来通知客户端代币被消费
    event Burn(address indexed from, uint256 value);

    /**
     * 初始化构造
     */
    function TokenERC20(uint256 initialSupply, string tokenName, string
tokenSymbol) public {
        totalSupply = initialSupply * 10 ** uint256(decimals);
        // 供应的份额，份额跟最小的代币单位有关，份额 = 币数 * 10 ** decimals
        balanceOf[msg.sender] = totalSupply;            // 创建者拥有所有的代币
        name = tokenName;                               // 代币名称
        symbol = tokenSymbol;                           // 代币符号
    }
```

```
/**
 * 代币交易转移的内部实现
 */
function _transfer(address _from, address _to, uint _value) internal {
    // 确保目标地址不为 0x0，因为 0x0 地址代表销毁
    require(_to != 0x0);
    // 检查发送者余额
    require(balanceOf[_from] >= _value);
    // 确保转移为正数个
    require(balanceOf[_to] + _value > balanceOf[_to]);

    // 以下用来检查交易
    uint previousBalances = balanceOf[_from] + balanceOf[_to];
    // Subtract from the sender
    balanceOf[_from] -= _value;
    // Add the same to the recipient
    balanceOf[_to] += _value;
    Transfer(_from, _to, _value);

    // 用 assert 来检查代码逻辑
    assert(balanceOf[_from] + balanceOf[_to] == previousBalances);
}

/**
 *  代币交易转移
 * 从自己（创建交易者）账号发送“_value”个代币到“_to”账号
 *
 * @param _to 接收者地址
 * @param _value 转移数额
 */
function transfer(address _to, uint256 _value) public {
    _transfer(msg.sender, _to, _value);
}

/**
 * 账号之间代币交易转移
 * @param _from 发送者地址
 * @param _to 接收者地址
```

```
     * @param _value 转移数额
     */
    function transferFrom(address _from, address _to, uint256 _value) public
returns (bool success) {
        require(_value <= allowance[_from][msg.sender]);     // Check allowance
        allowance[_from][msg.sender] -= _value;
        _transfer(_from, _to, _value);
        return true;
    }

    /**
     * 设置某个地址（合约）可以创建交易者名义花费的代币数
     *
     * 允许发送者“_spender”花费不多于“_value”个代币
     *
     * @param _spender The address authorized to spend
     * @param _value the max amount they can spend
     */
    function approve(address _spender, uint256 _value) public
        returns (bool success) {
        allowance[msg.sender][_spender] = _value;
        return true;
    }

    /**
     * 设置允许一个地址（合约）以我（创建交易者）的名义可最多花费的代币数
     *
     * @param _spender 被授权的地址（合约）
     * @param _value 最大可花费代币数
     * @param _extraData 发送给合约的附加数据
     */
    function approveAndCall(address _spender, uint256 _value, bytes _extraData)
        public
        returns (bool success) {
        tokenRecipient spender = tokenRecipient(_spender);
        if (approve(_spender, _value)) {
            // 通知合约
            spender.receiveApproval(msg.sender, _value, this, _extraData);
```

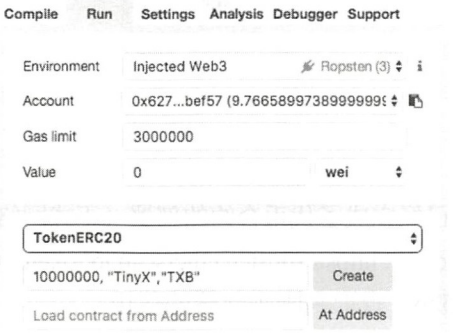
```
            return true;
        }
    }

    /**
     * 销毁我（创建交易者）账户中指定个数的代币
     */
    function burn(uint256 _value) public returns (bool success) {
        require(balanceOf[msg.sender] >= _value);   // Check if the sender has
                                                    // enough
        balanceOf[msg.sender] -= _value;            // Subtract from the sender
        totalSupply -= _value;                      // Updates totalSupply
        Burn(msg.sender, _value);
        return true;
    }

    /**
     * 销毁用户账户中指定个数的代币
     *
     * Remove "_value" tokens from the system irreversibly on behalf of "_from".
     *
     * @param _from the address of the sender
     * @param _value the amount of money to burn
     */
    function burnFrom(address _from, uint256 _value) public returns (bool success)
{
        require(balanceOf[_from] >= _value);      // Check if the targeted
                                                  // balance is enough
        require(_value <= allowance[_from][msg.sender]);    // Check allowance
        balanceOf[_from] -= _value;         // Subtract from the targeted balance
        allowance[_from][msg.sender] -= _value;  // Subtract from the sender's
                                                 // allowance
        totalSupply -= _value;                  // Update totalSupply
        Burn(_from, _value);
        return true;
    }
}
```

可以去 GitHub（https://github.com/xilibi2003/TokenERC20）查看代码。

部署

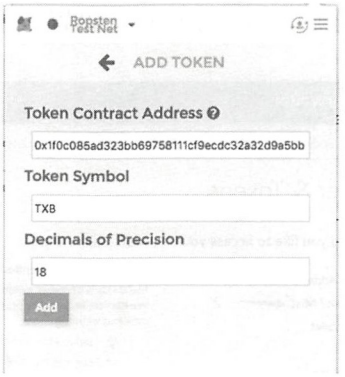
在开发测试智能合约时，MetaMask 和 Remix 是两个非常好用的工具，安装和配置 MetaMask 请参考第 9 章合约部署。部署的步骤如下：

在浏览器中打开 Remix Solidity IDE，复制并粘贴以上源码，设置如下图所示。

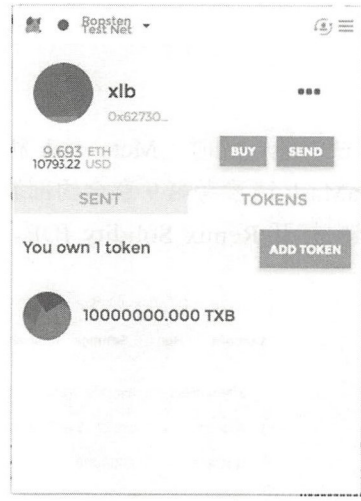


注意 Environment 和 Account、MetaMask 保持一致，然后选择合约 TokenERC20，填入你想要的发行量、名称及代号。这时 MetaMask 会弹出一个交易确认框，点击 SUBMIT 按钮。待合约部署交易确认之后，复制合约地址。

打开 Metamask 界面，切换到 TOKENS，添加合约，出现如下对话框。



填入刚刚复制的地址，点击 Add 按钮，这时你就可以看到新创建的代币了，如下图所示。

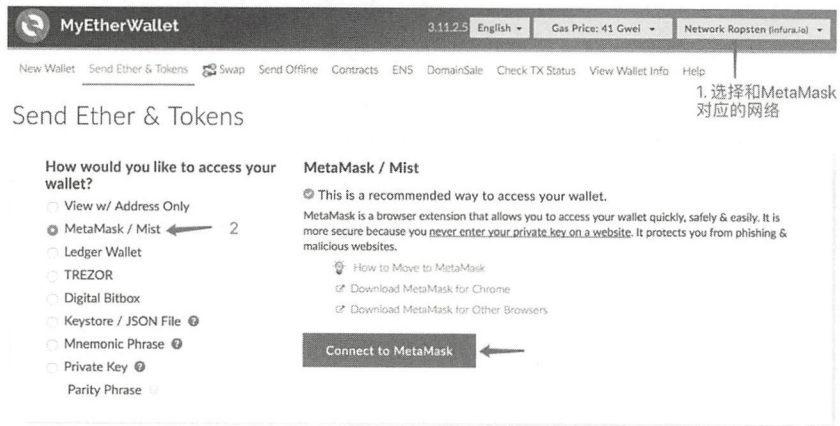


现在已经完成了代币合约的部署（正式网络和测试网络部署方法一样），可以在 Etherscan（https://ropsten.etherscan.io/token/ 0x1f0c085ad323bb69758111cf9ecdc32a32d9a5bb）查询到刚刚部署的代币。

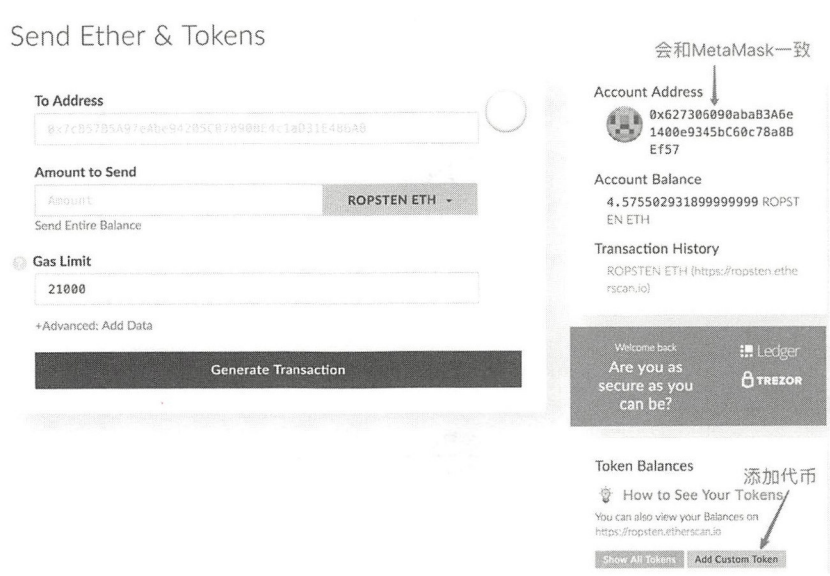
代币交易

由于 MetaMask 插件没有提供代币交易功能，同时考虑到很多人并没有以太坊钱包或是被以太坊钱包网络同步问题折磨，这里使用网页钱包（https://www.myetherwallet.com）来讲解代币交易。

1. 进入网页钱包，第一次进入有一些安全提示需要用户确认。
2. 进入之后，按照下图进行设置。

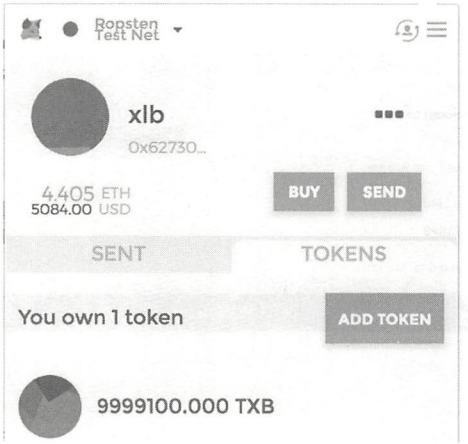


3．连接上之后，如下图所示。



4．需要添加代币，填入代币合约地址，进行代币转账交易。

5. 接下来进行交易确认，点击确认即可。交易完成后，可以看到 MetaMask 中代币余额减少了，如下图所示。



高级功能代币

代币是智能合约中最常见的合约，很多时候大家对它也有额外的需求。例如，如何进行代币管理、代币增发、代币兑换、资产冻结、gas 自动补充等，这一节我们就来介绍一下。

实现代币的管理者

虽然区块链是去中心化的，但是为了对代币进行管理，首先需要给合约添加一个管理者。

我们来看看这是如何实现的，先创建一个 owned 合约。

```
contract owned {
    address public owner;

    function owned() {
        owner = msg.sender;
    }

    modifier onlyOwner {
        require(msg.sender == owner);
        _;
```


```
    }

    // 实现所有权转移
    function transferOwnership(address newOwner) onlyOwner {
        owner = newOwner;
    }
}
```

这个合约中最重要的部分是加入了一个函数修改器 onlyOwner。函数修改器是一个合约属性，可以被继承，还能被重写。它用于在函数执行前检查某种前置条件。

> 熟悉 Python 的读者会发现函数修改器的作用和 Python 的装饰器很相似。

然后让代币合约继承 owned，以拥有 onlyOwner 修改器，代码如下：

```
contract MyToken is owned {
    function MyToken(
        uint256 initialSupply,
        string tokenName,
        uint8 decimalUnits,
        string tokenSymbol,
        address centralMinter
        ) {
        if(centralMinter != 0 ) owner = centralMinter;
    }
}
```

代币增发

实现代币增发。代币增发就如同央行印钞票一样，想必很多人都需要这样的功能。给合约添加以下方法：

```
function mintToken(address target, uint256 mintedAmount) onlyOwner {
        balanceOf[target] += mintedAmount;
        totalSupply += mintedAmount;
        Transfer(0, owner, mintedAmount);
        Transfer(owner, target, mintedAmount);
```


```
    }
```

注意 onlyOwner 修改器添加在函数末尾，这表示只有 ower 才能调用这个函数。功能很简单，就是给指定的账户增加代币，同时增加总供应量。

资产冻结

有时为了监管的需要，需要冻结某些账户。冻结后，其资产仍在账户中，但是不允许交易，之后解除冻结。给合约添加以下变量和方法（可以添加到合约的任何地方，但是建议把 mapping 和其他 mapping 放一起，event 也是如此）。

```
    mapping (address => bool) public frozenAccount;
    event FrozenFunds(address target, bool frozen);

    function freezeAccount(address target, bool freeze) onlyOwner {
        frozenAccount[target] = freeze;
        FrozenFunds(target, freeze);
    }
```

仅用以上代码还无法冻结，需要加入 transfer 函数才能真正生效，因此必须修改 transfer 函数。

```
function transfer(address _to, uint256 _value) {
        require(!frozenAccount[msg.sender]);
        ...
}
```

在转账前，对发起交易的账号做一次检查，只有未被冻结的账号才能转账。

代币买卖（兑换）

在自己的货币中实现代币与其他数字货币（Ether 或其他 Token）的兑换机制。有了这个功能，我们的合约就可以在一买一卖中赚利润了。

先来设置一下买卖价格，如下所示。

```
    uint256 public sellPrice;
    uint256 public buyPrice;

    function setPrices(uint256 newSellPrice, uint256 newBuyPrice) onlyOwner {
        sellPrice = newSellPrice;
```


```
        buyPrice = newBuyPrice;
    }
```

setPrices()添加了 onlyOwner 修改器，注意买卖的价格单位是 wei（最小的货币单位：1 eth = 1000000000000000000 wei)

买卖函数如下：

```
    function buy() payable returns (uint amount){
        amount = msg.value / buyPrice;                    // calculates the amount
        require(balanceOf[this] >= amount);               // 检查余额
        balanceOf[msg.sender] += amount;     // adds the amount to buyer's balance
        balanceOf[this] -= amount;       // subtracts amount from seller's balance
        Transfer(this, msg.sender, amount);        // execute an event reflecting
                                                   // the change
        return amount;                              // ends function and returns
    }

    function sell(uint amount) returns (uint revenue){
        require(balanceOf[msg.sender] >= amount);      // checks if the sender has
                                                       // enough to sell
        balanceOf[this] += amount;         // adds the amount to owner's balance
        balanceOf[msg.sender] -= amount;               // subtracts the amount
                                                       // from seller's balance
        revenue = amount * sellPrice;
        msg.sender.transfer(revenue);     // sends ether to the seller: it's
                                          // important to do this last to prevent
                                          // recursion attacks
        Transfer(msg.sender, this, amount);     // executes an event reflecting
                                                // on the change
        return revenue;                    // ends function and returns
    }
```

加入了买卖功能后，要求我们在创建合约时发送足够多的以太币，以便合约有能力回购市面上的代币，否则合约破产。

实现 gas 的自动补充

在以太坊中交易时需要 gas（支付给矿工的费用，费用以 Ether 支付）。如果用户没有以太币，那么在只有代币的情况下（或者我们想向用户隐藏以太坊

的细节），就需要自动补充 gas 的功能。这个功能将使我们的代币更加好用。

自动补充的逻辑是这样的：在执行交易之前，我们预先判断用户的余额（用来支付矿工的费用），如果用户的余额非常少（低于某个阈值时）可能影响到交易进行，那么合约会自动售出一部分代币来补充余额，以帮助用户顺利完成交易。

先来设定余额阈值：

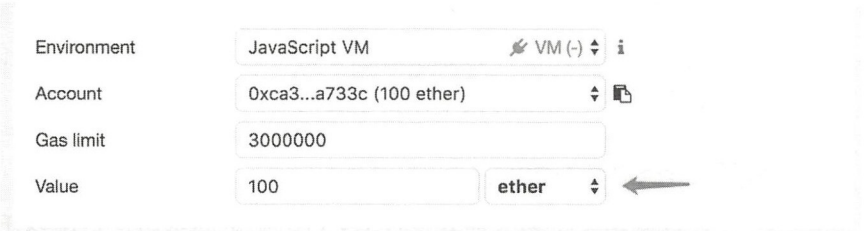
```
uint minBalanceForAccounts;

    function setMinBalance(uint minimumBalanceInFinney) onlyOwner {
         minBalanceForAccounts = minimumBalanceInFinney * 1 finney;
}
```

finney 是货币单位 1 finney = 0.001eth。我们在交易中加入对用户余额的判断：

```
   function transfer(address _to, uint256 _value) {
        ...
        if(msg.sender.balance < minBalanceForAccounts)
            sell((minBalanceForAccounts - msg.sender.balance) / sellPrice);
        if(_to.balance<minBalanceForAccounts)    // 可选，让接受者也补充余额
                                                 // 以便接受者使用代币
            _to.send(sell((minBalanceForAccounts - _to.balance) / sellPrice));
    }
```

代码部署

高级功能完整代码已经上传到 GitHub（https://github.com/xilibi2003/TokenERC20）上。项目完整的部署方法参考上一个案例。不同的是，创建合约时需要预存余额，如下图所示。



众筹（ICO）合约

使用代币来公开募资，众筹合约是数字货币世界里最常见的合约之一。我们来看看它是如何实现的。

众筹

先简单说一下“众筹”的概念。一般是这样的，例如我有一个非常好的想法，但是我没有钱来做这件事，于是我把这个想法发给大家看一下，然后说：“我做这件事需要 500 万元，大家有没有兴趣投些钱。如果大家在 30 天内投够了 500 万元我就开始做，到时大家都是原始股东，如果募资额不到 500 万元，大家投的钱就还给大家。”

区块链技术本来就非常适合解决众筹的信任问题。借助智能合约，可以实现当募资额完成时，募资款自动打到指定账户的功能；当募资额未完成时，可退款。这个过程不需要看众筹发起者的人品，不用依靠第三方平台的信用担保。

代币

传统的众筹在参与之后通常不容易交易（参与之后无法转给其他人），而通过用代币来参与众筹，则很容易进行交易。众筹的参与人可随时进行买卖，待众筹项目实施完成时，可根据代币持有量进行回馈。

举例说明，大家会更容易理解。例如有这样一个众筹：A 有技术做一个能监测健康的指环，为此向公众募资 200 万元。募资时 100 元对应一个代币。约定在指环上市之后，代币的持有人可以用一个代币来兑换一个指环。而指环的研发周期是一年，因此在指环还未上市的一年里，众筹的参与人可以随时交易所持有的代币。

众筹智能合约代码

我们来看看如何实现这个众筹智能合约的，如下所示。

```
pragma solidity ^0.4.16;

interface token {
    function transfer(address receiver, uint amount);
}
```

```
contract Crowdsale {
    address public beneficiary;  // 募资成功后的收款方
    uint public fundingGoal;   // 募资额度
    uint public amountRaised;   // 参与数量
    uint public deadline;      // 募资截止日期

    uint public price;    //  token 与以太坊的汇率 , token 卖多少钱
    token public tokenReward;   // 要卖的 token

    mapping(address => uint256) public balanceOf;

    bool fundingGoalReached = false;  // 众筹是否达到目标
    bool crowdsaleClosed = false;   //  众筹是否结束

    /**
    * 事件可以用来跟踪信息
    **/
    event GoalReached(address recipient, uint totalAmountRaised);
    event FundTransfer(address backer, uint amount, bool isContribution);

    /**
     * 构造函数, 设置相关属性
     */
    function Crowdsale(
        address ifSuccessfulSendTo,
        uint fundingGoalInEthers,
        uint durationInMinutes,
        uint finneyCostOfEachToken,
        address addressOfTokenUsedAsReward) {
            beneficiary = ifSuccessfulSendTo;
            fundingGoal = fundingGoalInEthers * 1 ether;
            deadline = now + durationInMinutes * 1 minutes;
            price = finneyCostOfEachToken * 1 finney;
            tokenReward = token(addressOfTokenUsedAsReward);
           // 传入已发布的 token 合约的地址来创建实例
    }

    /**
```

```
 * 无函数名的 Fallback 函数
 * 在向合约转账时，这个函数会被调用
 */
function () payable {
    require(!crowdsaleClosed);
    uint amount = msg.value;
    balanceOf[msg.sender] += amount;
    amountRaised += amount;
    tokenReward.transfer(msg.sender, amount / price);
    FundTransfer(msg.sender, amount, true);
}

/**
*  定义函数修改器 modifier（作用和 Python 的装饰器很相似）
* 用于在函数执行前检查某种前置条件（判断通过之后才会继续执行该方法）
* _ 表示继续执行之后的代码
**/
modifier afterDeadline() { if (now >= deadline) _; }

/**
 * 判断众筹是否完成融资目标，这个方法使用了 afterDeadline 函数修改器
 *
 */
function checkGoalReached() afterDeadline {
    if (amountRaised >= fundingGoal) {
        fundingGoalReached = true;
        GoalReached(beneficiary, amountRaised);
    }
    crowdsaleClosed = true;
}


/**
 * 完成融资目标时，融资款发送到收款方
 * 未完成融资目标时，执行退款
 *
 */
function safeWithdrawal() afterDeadline {
```

```
        if (!fundingGoalReached) {
            uint amount = balanceOf[msg.sender];
            balanceOf[msg.sender] = 0;
            if (amount > 0) {
                if (msg.sender.send(amount)) {
                    FundTransfer(msg.sender, amount, false);
                } else {
                    balanceOf[msg.sender] = amount;
                }
            }
        }

        if (fundingGoalReached && beneficiary == msg.sender) {
            if (beneficiary.send(amountRaised)) {
                FundTransfer(beneficiary, amountRaised, false);
            } else {
                fundingGoalReached = false;
            }
        }
    }
}
```

部署及说明

在部署这个合约之前，我们需要先部署一个代币合约。

创建众筹合约我们需要提供以下几个参数。

ifSuccessfulSendTo：募资成功后的收款方（这里可以默认为合约创建者）。

fundingGoalInEthers：募资额度，即募资多少以太币。

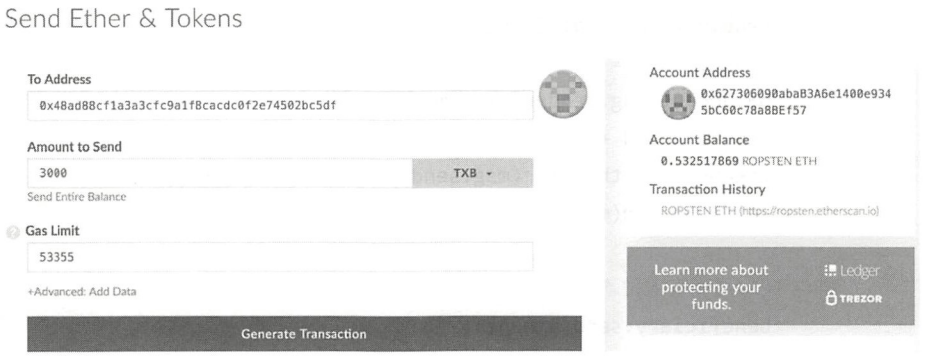
durationInMinutes：募资时间（可以根据需要修改为“天”）。

finneyCostOfEachToken：每个代币的价格，每个代币需要多少 finney 可以根据需要修改为其他货币单位。

addressOfTokenUsedAsReward：代币合约地址。

如下图所示。

参与人投资的时候实际购买众筹合约代币，所以需要先向众筹合约预存代币，代币的数量为：募资额度 / 代币的价格。向合约预存代币时可以使用任何钱包进行代币转账，转账的地址就是我们创建合约的地址（也可在 Remix 中加载代币合约，执行合约 tranfer()函数进行转账）。例如使用 myetherwallet 转账，如下图所示。



1．投资人向众筹合约转账（发送以太币）即参与众筹。转账时，会执行众筹合约回退函数，向其账户打回相应的代币。

2．safeWithdrawl() 可以被参与人或收益人调用，如果融资不达标，那么参与人可收回之前的投资款；如果融资达标，那么收益人提取所有的融资款。

无限众筹合约

以上是一个很正规的众筹合约。在上面的合约中，众筹额度和相应的代币都是固定的，一旦达到众筹目标后，就无法再继续投资了。如果想实现一个众筹达标后仍然可以继续募资的合约，则操作步骤如下。

1．先参考高级工程代币章节，部署一个可增发代币合约。

2．更改回退函数。

```
tokenReward.transfer(msg.sender, amount / price);
//修改为：
tokenReward.mintToken(msg.sender, amount / price);
```

3．在无限众筹合约部署之后，调用高级代币合约的 transferOwnership。把无限众筹合约地址设置为代币合约的 Owner，以便众筹合约能执行 mintToken() 函数。

“割韭菜”合约

在上面的合约中，如果募资未达标，则参与投资的人能够取回自己的投资。这个逻辑也是可以更改的，如下所示。

```
function () payable {
  require(!crowdsaleClosed);
  uint amount = msg.value;
  balanceOf[msg.sender] += amount;
  amountRaised += amount;
  tokenReward.transfer(msg.sender, amount / price);
  FundTransfer(msg.sender, amount, true);

    // 当有人付款，则直接取走资金
  beneficiary.send(amount);

}
```

非同质化代币 ERC721

什么是 ERC-721？现在我们看到的各种加密猫游戏，基本都是基于 ERC721 创造出来的，每只猫都是一个独一无二的 ERC721 代币。不过 ERC721 在区块链世界中远不止猫猫狗狗，对它更大的想象空间在于将物理世界的资产映射到区块链上，本节我们就来剖析一下什么是 ERC721。

ERC721 是什么

前面我们介绍了 ERC20 代币。和 ERC20 一样，ERC721 同样是一个代币标准，ERC721 的官方解释是 Non-Fungible Tokens，简写为 NFTs，翻译为“非同质代币”。

> ERC721 是 Dieter Shirley 在 2017 年 9 月提出的。Dieter Shirley 正是“谜恋猫”（CryptoKitties）背后的公司 Axiom Zen 的技术总监。因此谜恋猫也是第一个实现了 ERC721 标准的去中心化应用。ERC721 已经被以太坊作为标准接受，但该标准仍处于草稿阶段。本文介绍的 ERC721 标准基于最新（2018/03/23）官方提议。

那么怎么理解**非同质**代币呢?

“非同质”代表独一无二。以谜恋猫为例，每只猫都被赋予了基因，是独一无二的（一只猫就是一个 NFTs），猫之间是不能置换的。这种独特性使得某些稀有猫具有收藏价值，也因此受到追捧。

ERC20 代币是可置换的，并且可细分为 N 份（1 = 10 * 0.1），而 ERC721 的 Token 最小的单位为 1，无法再分割。

> 如果同一个集合的两个物品具有不同的特征，那么这两个物品是非同质的。而同质是指某个部分或数量可以被另一个同等部分或数量所代替。

非同质性其实广泛存在于我们的生活中，例如图书馆的每一本书、宠物商店的每一只宠物、歌手所演唱的歌曲、花店里不同的花，等等。因此 ERC721 合约必定有广泛的应用场景。通过这样一个标准，也可建立跨功能的 NFTs 管理和销售平台（就像所有支持 ERC20 的交易所和钱包一样），使生态更加强大。

ERC721 标准

ERC721 是一个合约标准，它提供了在实现 ERC721 代币时必须要遵守的协议，并且要求每个 ERC721 标准合约需要实现 ERC721 及 ERC165 接口，接口定义如下所示。

```
pragma solidity ^0.4.20;

interface ERC721 /* is ERC165 */ {
```

```
    event Transfer(address indexed _from, address indexed _to, uint256 _tokenId);
    event Approval(address indexed _owner, address indexed _approved, uint256
_tokenId);
    event ApprovalForAll(address indexed _owner, address indexed _operator, bool
_approved);

    function balanceOf(address _owner) external view returns (uint256);
    function ownerOf(uint256 _tokenId) external view returns (address);

    function safeTransferFrom(address _from, address _to, uint256 _tokenId,
bytes data) external payable;
    function safeTransferFrom(address _from, address _to, uint256 _tokenId)
external payable;
    function transferFrom(address _from, address _to, uint256 _tokenId) external
payable;

    function approve(address _approved, uint256 _tokenId) external payable;
    function setApprovalForAll(address _operator, bool _approved) external;
    function getApproved(uint256 _tokenId) external view returns (address);
    function isApprovedForAll(address _owner, address _operator) external view
returns (bool);
}
```

以下为接口说明。

- balanceOf()：返回由_owner 持有的 NFTs 的数量。
- ownerOf()：返回 tokenId 代币持有者的地址。
- approve()：授予地址 to 具有 tokenId 的控制权，方法成功后需触发 Approval 事件。
- setApprovalForAll：授予地址_operator 具有所有 NFTs 的控制权，成功后需触发 ApprovalForAll 事件。
- getApproved()、isApprovedForAll()：用来查询授权。

safeTransferFrom()：转移 NFT 所有权，一次成功的转移操作必须发起 Transer 事件。

函数的实现需要做以下几种检查：

- 调用者 msg.sender 应该是当前 tokenId 的所有者或被授权的地址。

- _from 必须是_tokenId 的所有者。
- _tokenId 应该是当前合约正在监测的 NFTs 中的任何一个。
- _to 地址不应该为 0。
- 如果_to 是一个合约，则应该调用其 onERC721Received 方法，并且检查其返回值。如果返回值不为 bytes4(keccak256("onERC721Received(address,uint256,bytes)"))，则抛出异常。一个可接收 NFT 的合约必须实现 ERC721TokenReceiver 接口。

```
interface ERC721TokenReceiver {
    // @return bytes4(keccak256("onERC721Received(address, uint256,bytes)"))
    function onERC721Received(address _from, uint256 _tokenId, bytes data)
external returns(bytes4);
}
```

- transferFrom()：用来转移 NFTs，方法成功后需触发 Transfer 事件。调用者自己确认_to 地址能否正常接收 NFT，否则将丢失此 NFT。此函数实现时需要检查上面条件中的前 4 条。

ERC165 标准

ERC721 标准同时要求必须符合 ERC165 标准 ，其接口如下所示。

```
interface ERC165 {
    function supportsInterface(bytes4 interfaceID) external view returns (bool);
}
```

ERC165 同样是一个合约标准。这个标准要求合约提供其实现了哪些接口，这样再与合约进行交互的时候可以先调用此接口进行查询。interfaceID 为函数选择器，计算方式有两种，例如：bytes4(keccak256 ('supportsInterface (bytes4)')); 或 ERC165.supportsInterface.selector，多个函数的接口 ID 为函数选择器的异或值。

可选实现接口：ERC721Metadata

ERC721Metadata 接口用于提供合约的元数据 name、symbol 及 URI（NFT 所对应的资源）。其接口定义如下所示。

```
interface ERC721Metadata /* is ERC721 */ {
    function name() external pure returns (string _name);
```

```
    function symbol() external pure returns (string _symbol);
    function tokenURI(uint256 _tokenId) external view returns (string);
}
```

以下为接口说明。

- name()：返回合约名字。
- symbol()：返回合约代币符号。
- tokenURI()：返回_tokenId 所对应的外部资源文件的 URI（通常是 IPFS 或 HTTP(S)路径）。外部资源文件需要包含名字、描述、图片，其格式的要求如下所示。

```
{
    "title": "Asset Metadata",
    "type": "object",
    "properties": {
        "name": {
            "type": "string",
            "description": "Identifies the asset to which this NFT represents",
        },
        "description": {
            "type": "string",
            "description": "Describes the asset to which this NFT represents",
        },
        "image": {
            "type": "string",
            "description": "A URI pointing to a resource with mime type image/*
representing the asset to which this NFT represents. Consider making any images
at a width between 320 and 1080 pixels and aspect ratio between 1.91:1 and 4:5
inclusive.",
        }
    }
}
```

tokenURI 通常是被 Web3 调用，以便在应用层做相应的查询和展示。

可选实现接口：ERC721Enumerable

ERC721Enumerable 的主要目的是提高合约中 NTF 的可访问性，其接口定

义如下所示。

```
interface ERC721Enumerable /* is ERC721 */ {
    function totalSupply() external view returns (uint256);
    function tokenByIndex(uint256 _index) external view returns (uint256);
    function tokenOfOwnerByIndex(address _owner, uint256 _index) external view
returns (uint256);
}
```

以下为接口说明。

- totalSupply()：返回 NFT 总量。
- tokenByIndex()：通过索引返回对应的 tokenId。
- tokenOfOwnerByIndex()：所有者可以一次拥有多个 NFT，函数返回 _owner 拥有的 NFT 列表中对应索引的 tokenId。

补充说明

NTF ID

NTF ID 即 tokenId，在合约中用唯一的 uint265 进行标识。每个 NFT 的 ID 在智能合约的生命周期内不允许改变。推荐的实现方式有：

1．从 0 开始，每新加一个 NFT，NTF ID 加 1。

2．使用 sha3 后 uuid 转换为 NTF ID。

与 ERC20 的兼容性

ERC721 标准尽可能遵循 ERC20 的语义，但由于同质代币与非同质代币之间的根本差异，并不能完全兼容 ERC20。

交易、挖矿、销毁

在实现 transter 相关接口时除满足上面的条件外，我们也可以根据需要添加自己的逻辑，如加入黑名单等。另外，尽管挖矿、销毁不是标准的一部分，我们也可以根据需要实现。

本章小结

本章在编写时参考了以下文章：

- https://ethereum.org/token
- https://theethereum.wiki/w/index.php/ERC20_Token_Standard

- https://ethereum.org/crowdsale
- https://github.com/ethereum/EIPs/blob/master/EIPS/eip-721.md
- https://github.com/ethereum/EIPs/blob/master/EIPS/eip-165.md
- https://ethfiddle.com/09YbyJRfiI
- https://github.com/fulldecent/erc721-example
  https://github.com/nastassiasachs/ERC721ExampleDeed

希望本章案例能给读者带来一些智能合约编写的启发。

第 13 章

去中心化应用开发

现在的互联网应用通常都有相应的中心化服务器，在应用端（前端）展现内容的时候，通常发送一个请求到服务器，服务器返回相应的内容给应用端。在应用端的动作也是一样会转化请求发送到服务端，服务器处理之后返回数据到应用端。整个应用实际是由中心化的服务器控制的。

DApp（Decentralized App，去中心化应用），则是运行在去中心化的网络节点上的，应用端其实和现有互联网应用一样，不过应用的后端不再是中心化的服务器，而是去中心化的网络中的节点。这个节点可以是网络中任意的节点，应用端发给节点的请求，节点收到交易请求之后，会把请求广播到整个网络，交易在网络达成共识之后，才算是真正的执行（即处理请求的是整个网络，连接的节点不能独立处理请求）。

在去中心化应用中，发送给节点的请求通常称为“交易”。交易和普通的请求有很大的不同，即交易的数据经过用户个人签名（因此需要关联钱包）之后发送到节点，另外普通的请求大多数都是同步的（及时拿到结果），而交易大多数都是异步的（因为网络共识比较耗时），还有交易不是使用普通 HTTP JSON 请求，而是使用 JSON RPC 请求。

本章将介绍如何基于以太坊来编写去中心化应用。

JSON RPC

JSON RPC 介绍

JSON RPC（Remote Protocol Call）是一种以 JSON 为格式的远程调用协议，其请求和返回都是 JSON 格式，常见的请求格式如下：

```
{
    "jsonrpc" : 2.0,
    "method" : "echoHello",
    "params" : ["Hello JSON-RPC"],
    "id" : 1
}
```

jsonrpc 定义了 JSON RPC 版本。method 为调用方法名。params 为传入的参数，若无参数则为 null。id 为调用标识符，可以为字符串。

返回也是 JSON 格式：

```
{
    "jsonrpc" : 2.0,
    "result" : "Hello JSON-RPC",
    "error" : null,
    "id" : 1
}
```

jsonrpc：定义 JSON RPC 版本。

result：方法返回值。error 调用时错误，无错误时返回 null，有错误时则返回一个错误对象。

id：调用标识符，与调用方传入的标识一致，当请求中的 id 检查发生错误时（转换错误 / 无效请求），则必须返回 null。

关于 JSON RPC 的详细规范可以查阅说明文档（http://www.jsonrpc.org/specification）。

如何与以太坊节点进行通信

以太坊使用 JSON RPC 2.0 规范来和节点进行通信，我们来看看这个步骤是怎样实现的：

首先要求我们在启动节点时，加入--rpc 选项，如下所示。

```
geth --rpc
```

geth 会默认使用 8545 进行监听 JSON RPC 请求，如果要更改端口，使用 --rpcport <portnumber>向节点发送 JSON RPC 请求，这里以请求账户余额为例，方法为 eth_getBalance 进行说明。

```
curl -X POST --data '{"jsonrpc":"2.0","method":"eth_getBalance","params":
["0x407d73d8a49eeb85d32cf465507dd71d507100c1", "latest"],"id":1}' -H
"Content-Type: application/json" localhost:8545
// 返回
{
  "id":1,
  "jsonrpc": "2.0",
  "result": "0x0234c8a3397aab58" // 以 wei 为单位
}
```

在返回的结果中，可以从 result 里拿到余额，需要注意的是 JSON 里的数字是十六进制编码。

除 eth_getBalance 方法之外，常用的如发送交易（用于和合约互动或创建合约）的方法为 eth_sendTransaction，获取账号的方法为 eth_accounts，所有方法的使用可以通过官方文档：https://github.com/ethereum/wiki/wiki/JSON-RPC 查看。

Web3.js

虽然通过使用 JSON RPC 请求虽然可以完成和节点的通信，但是这个请求比较烦琐，因为需要和原始的底层数据交互，参数很多，返回的数据也需要自己解析，比较容易出错。Web3.js 是以太坊官方提供的和节点交互的 JavaScript SDK，可以帮助智能合约开发者使用 HTTP 或者 IPC 与本地的或者远程的以太坊节点进行交互。当然 Web3.js 同样是使用 JSON RPC 和节点进行通信的（Web3 是对 JSON RPC 请求的封装），不过 Web3.js 提供了更友好的接口，实际上 Web3.js 就是一个库的集合，主要包括下面几个库：

- web3-eth 用来与以太坊区块链和智能合约交互。
- web3-shh 用来控制 whisper 协议与 p2p 通信，以及广播。
- web3-bzz 用来与 swarm 协议交互。

- web3-utils 包含了一些 DApp 开发应用的功能。

在 geth 中使用 Web3.js

geth 启动的时候会自动加载 Web3.js 库，因此可以在 geth 交互控制台里直接使用 Web3.js。这在第 9 章介绍合约部署的时候已经使用过，下面是一个接口使用示例：

```
> eth.accounts
> personal.newAccount("accountName")
> eth.getBalance("accountAddr")
```

通常我们在开发的时候，可以在 geth 控制台中进行一些接口的测试及验证。Web3.js API 具体提供了哪些接口可以参看文档，目前有两个不兼容的版本：

http://web3js.readthedocs.io/en/1.0/index.html

https://github.com/ethereum/wiki/wiki/JavaScript-API

大家也可以阅读我翻译的版本：https://web3.learnblockchain.cn。

在应用中使用 Web3.js

这是最常用的 Web3 使用方式，在我们开发的应用中需要引入 Web3.js 库来获取节点账号等信息，以及和智能合约进行交互。

项目引入 Web3.js

首先你需要将 Web3 引入到工程中，根据项目的不同，使用不同的方式。

- npm，npm install Web3。
- bower，bower install Web3。
- meteor，meteor add ethereum:Web3。
- vanilla，dist./Web3.min.js。

创建 Web3 实例

然后提供一个 Provider 来创建一个 Web3 的实例，为了不覆盖一个已有的 Provider，需要先检查 Web3 实例是否已存在。因为在 Mist 中，在有 MetaMask 插件的浏览器中使用时会提供 Provider。

创建实例的方法如下：

```
if (typeof web3 !== 'undefined') {
  web3 = new Web3(web3.currentProvider);
} else {
  // set the provider you want from Web3.providers
  web3 = new Web3(new Web3.providers.HttpProvider("http://localhost:8545"));
}
```

创建好 Web3 对象后，就可以使用 Web3.js 提供的 API 了。Web3.js 所有的 API 可以在网站（http://web3js.readthedocs.io/en/1.0/index.html）上查询。

使用回调

如果使用 Web3.js API 用来与本地的 RPC 结点交互，函数默认使用同步的 HTTP 请求，否则应该使用异步请求。

如果想发起一个异步的请求，那么大多数函数允许传一个跟在参数列表后的可选的回调函数来支持异步。回调函数支持错误优先的回调模式（Error First Callback）。例如：

```
web3.eth.getBlock(48, function(error, result){
    if(!error)
        console.log(result)
    else
        console.error(error);
})
```

批量请求

可以允许将多个请求放入队列并一起执行，方法如下所示。

```
var batch = web3.createBatch();
batch.add(web3.eth.getBalance.request('0x0000000000000000000000000000000000000
00000', 'latest', callback));
batch.add(web3.eth.Contract(abi).at(address).balance.request(address,
callback2));
batch.execute();
```

> 注意：批量请求并不会更快，批量请求的主要目的是用来保证请求的串行执行。实际上同时发起多个请求会更快，因为请求是异步处理的。

处理大数据

数据类型的返回结果，得到一个 BigNumber 对象，因为 JavaScript 不能正确地处理 BigNumber，如下所示。

```
"10101010032432534534645645645645645645"
// "101010100324325345346456456456456456456"
1010101003243253453464564564564564564
// 1.0101010032432535e+38
```

所以 Web3.js 依赖 BigNumber 库，并且会自动进行引入，如下所示。

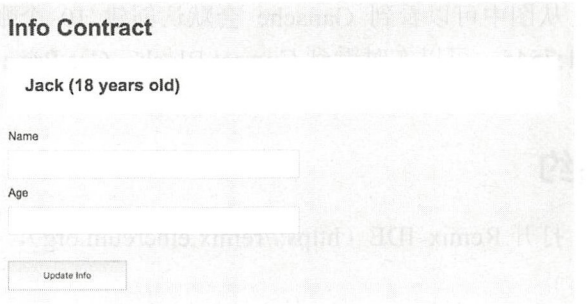
```
var balance = new BigNumber('131242344353464564564574574567456');
// 或 var balance = web3.eth.getBalance(someAddress);

balance.plus(21).toString(10); // toString(10) 转为一个数字字符串
// "131242344353464564564574574567477"
```

再看下面一个例子，即使有 20 位以上的浮点值，也会出错。所以，尽量让账户余额以 wei 为单位，仅仅在需要向用户展示时，才转换为其他单位。

```
var balance = new BigNumber('13124.2344353464564666664574555567456');

balance.plus(21).toString(10); // 数字字符串没有显示 20 位的小数
// "13145.23443534645646666646"
```

去中心化应用案例

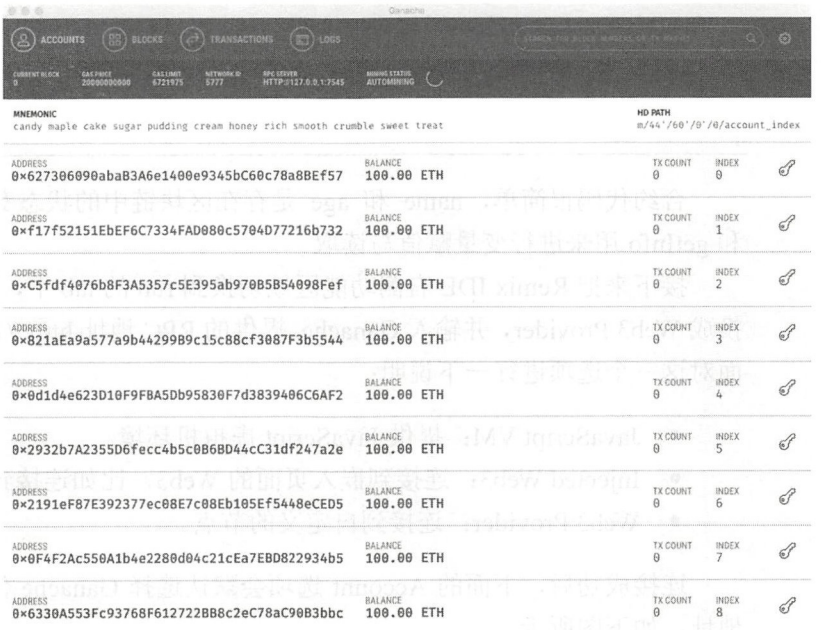
我们结合一个简单的案例来说明 Web3.js 在去中心化应用中的使用。下面是一个 Web 应用，其开发完成之后的界面，如下所示。



这个应用的名字（Name）和年龄（Age）数据是存储在区块链上的。我们来看看是如何实现的。

搭建测试环境

在开发初期，我们没有必要使用真实的公链。为了开发效率，一般选择在本地搭建环境。这里使用 Ganache（http://truffleframework.com/ganache/），它可以提供一个在内存模拟的以太坊节点环境（Ganache CLI 是对应的命令行版本），安装之后，打开的界面如下所示。



从图中可以看到 Ganache 会默认创建 10 个账户，监听地址是 http://127.0.0.1:7545，可以实时看到 Current Block、Gas Price、Gas Limit 等信息。

创建智能合约

打开 Remix IDE（https://remix.ethereum.org），在代码区域编写智能合约代码：

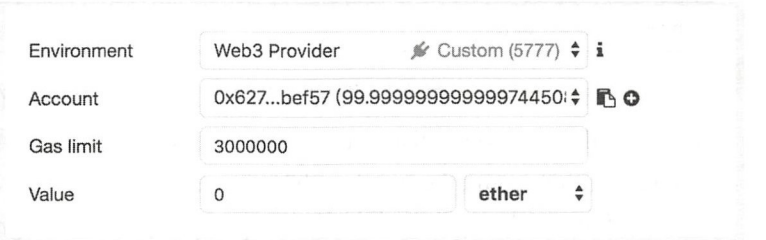
```
pragma solidity ^0.4.21;

contract InfoContract {

   string fName;
   uint age;

   function setInfo(string _fName, uint _age) public {
       fName = _fName;
       age = _age;
   }

   function getInfo() public constant returns (string, uint) {
       return (fName, age);
   }
}
```

合约代码很简单，name 和 age 是存在区块链中的状态变量，函数 setInfo 和 getInfo 用来进行变量赋值与读取。

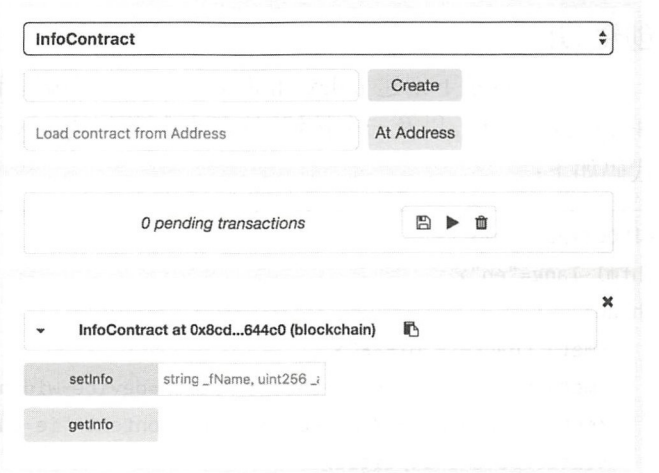
接下来把 Remix IDE 右侧功能区域切换到 run 的 tab 下，将 Environment 切换成 Web3 Provider，并输入 Ganache 提供的 RPC 地址 http://127.0.0.1:7545。下面对这三个选项进行一下说明：

- JavaScript VM：提供 JavaScript 虚拟机环境。
- Injected Web3：连接到嵌入页面的 Web3，比如连接到 MetaMask。
- Web3 Provider：连接到自定义的节点。

连接成功后，下面的 Account 选项会默认选择 Ganache 创建的第一个账户地址，如下图所示。



点击 Create，就会将智能合约部署到我们的测试环境中，如下图所示。



智能合约部署之后，接下来要编写应用 UI 及跟合约交互的部分。

创建项目，安装 Web3

先创建应用的项目目录，使用以下命令。

```
> mkdir info
> cd info
```

使用 node.js（安装 Node）的包管理工具 npm 初始化项目。

```
> npm init
```

在命令运行期间，输入项目名称、版本等信息。项目创建完成后，生成一个 package.json 文件，保存项目信息及相关依赖。

然后运行命令，安装 Web3.js。

```
> npm install web3
```

这里有个问题，在实际安装过程中，我发现 Web3 在安装完成后并没有/node_modules/ web3/dist/web3.min.js 文件。这个问题在 issue#1041（https://github.com/ethereum/ web3.js/issues/1041）有提到，但官方好像一直没解决。不过我们可以在这个链接：https://cdn.jsdelivr.net/gh/ethereum/web3.js/下载所需的文件，拷贝到/node_modules/web3 路径（其他路径也可以，下文 script 要注意和放置目录一致）下。

创建 UI

在项目目录下创建 index.html 文件，在这里编写基础的 UI。UI 包括 name 和 age 的输入框，以及一个按钮。这里也会引入 jQuery 进行一些交互，index.html 代码如下：

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <meta http-equiv="X-UA-Compatible" content="ie=edge">
    <title>Document</title>

    <link rel="stylesheet" type="text/css" href="main.css">

    <script src="./node_modules/web3/dist/web3.min.js"></script>

</head>
<body>
    <div class="container">

        <h1>Info Contract</h1>

        <h2 id="info"></h2>

        <label for="name" class="col-lg-2 control-label">Name</label>
        <input id="name" type="text">
```

```
        <label for="name" class="col-lg-2 control-label">Age</label>
        <input id="age" type="text">

        <button id="button">Update Info</button>


    </div>

    <script src="https://code.jquery.com/jquery-3.2.1.slim.min.js"></script>

    <script>
       // 接下来添加……
    </script>

</body>
</html>
```

接下来需要编写 main.css 文件，设定基本的样式。

```
body {
    background-color:#F0F0F0;
    padding: 2em;
    font-family: 'Raleway','Source Sans Pro', 'Arial';
}
.container {
    width: 50%;
    margin: 0 auto;
}
label {
    display:block;
    margin-bottom:10px;
}
input {
    padding:10px;
    width: 50%;
    margin-bottom: 1em;
}
button {
    margin: 2em 0;
```

```
    padding: 1em 4em;
    display:block;
}

#info {
    padding:1em;
    background-color:#fff;
    margin: 1em 0;
}
```

使用 Web3 与智能合约交互

UI 创建好之后，在<script>标签中间编写 Web3.js 的代码与智能合约交互，关键的步骤和代码如下。

- 创建 Web3 实例，并且与 Ganache 提供的测试环境连接。

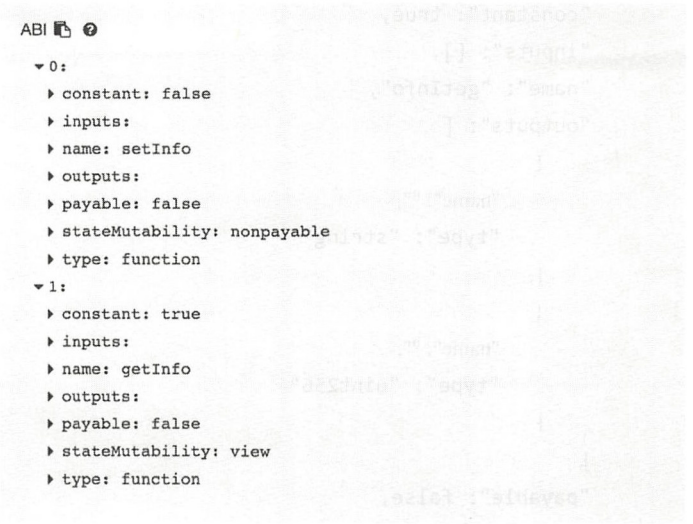
```
<script>
    if (typeof web3 !== 'undefined') {
        web3 = new Web3(web3.currentProvider);
    } else {
        web3 = new Web3(new Web3.providers.HttpProvider("http://
localhost:7545"));
    }
</script>
```

- 使用 Web3 API 设置默认的账户。Ganache 帮我们创建了 10 个账户，这里选择第一个账户当作默认账户。

```
web3.eth.defaultAccount = web3.eth.accounts[0];
```

- 创建合约实例。

这里需要用到合约的 ABI JSON，它包含一系列的函数或事件的描述。我们可以在 Compile 的 tab 下点击 Details，在出现的页面中拷贝合约的 ABI（点击“复制”小图标），如下图所示。



将其复制到代码中，代码如下所示。

```
// 复制 ABI 赋值给 abi
var abi = [
    {
        "constant": false,
        "inputs": [
            {
                "name": "_fName",
                "type": "string"
            },
            {
                "name": "_age",
                "type": "uint256"
            }
        ],
        "name": "setInfo",
        "outputs": [],
        "payable": false,
        "stateMutability": "nonpayable",
        "type": "function"
    },
    {
```

```
        "constant": true,
        "inputs": [],
        "name": "getInfo",
        "outputs": [
            {
                "name": "",
                "type": "string"
            },
            {
                "name": "",
                "type": "uint256"
            }
        ],
        "payable": false,
        "stateMutability": "view",
        "type": "function"
    }
]
var infoContract = web3.eth.contract(abi);
```

接着在 Remix Run 标签中把部署的合约地址也拷贝一下，将其复制到下面的代码中。

```
var info = InfoContract.at('合约地址');
```

这时就完成了合约实例的创建。

- 合约函数交互

使用合约实例调用合约中的函数。下面我们使用 jQuery 与合约进行交互。

```
info.getInfo(function(error, result){
    if(!error)
        {
            $("#info").html(result[0]+' ('+result[1]+' years old)');
            console.log(result);
        }
    else
        console.error(error);
});
```

```
$("#button").click(function() {
    info.setInfo($("#name").val(), $("#age").val());
});
```

以上代码实现了对合约中两个函数的调用，分别读取和显示 name 和 age 变量。到此我们就完成了应用的全部代码，完整代码可以在 GitHub（https://github.com/ xilibi2003/InfoContract）查看。

在浏览器中打开 index.html，输入名字和年龄后刷新一下，就可以看到前面贴出的效果图了。

合约加入事件

在上面的案例中，只有在刷新浏览器后才能看到数据的变化。如果要监听数据的变化，则需要在合约中发送事件。

先在合约中声明一个事件，如下所示。

```
event Instructor(
     string name,
     uint age
   );
```

在这个事件中会接收两个参数 name 和 age，也就是需要跟踪的两个信息。然后在 setInfo 函数中，触发 Instructor 事件，如下所示。

```
function setInfo(string _fName, uint _age) public {
    fName = _fName;
    age = _age;
    emit Instructor(_fName, _age);
}
```

使用 Web3 监听事件、刷新 UI

点击“Updata Info”按钮之后，会调用 setInfo 函数，触发 Instructor 事件。现在使用 Web3 监听事件，刷新 UI。在之前的 js 代码中，我们使用 info.getInfo() 来获取信息，现在我们改用监听事件获取信息。先定义一个变量引用事件，如

下所示。

```
var instructorEvent = info.Instructor();
```

然后使用.watch()方法来添加一个回调函数，如下所示。

```
instructorEvent.watch(function(error, result) {
       if (!error)
          {
             $("#info").html(result.args.name + ' (' + result.args.age + '
years old)');
          } else {
             console.log(error);
          }
    });
```

大家可以在我的 GitHub 上查看到完整的代码，地址为 https://github.com/xilibi2003/InfoContract。

Truffle 框架

上面我们介绍了如何开发一个简单的去中心化应用。但是上面的方式也有缺点，合约是单独部署管理，前端界面依赖合约的 ABI 及合约地址，这样合约每次进行更改的时候，前端界面也需要修改相应的代码，如果项目大一些，可能需要不停地进行智能合约编译、部署，对应的前端也需要不停地修改，这会让项目很难管理。这时就可以使用 Truffle 来进行开发了。Truffle 是目前最流行的以太坊开发框架，它可以帮我们简化开发流程，处理掉大量开发中的琐事，让我们可以迅速开始写代码—编译—部署—测试—打包 DApp 这一整个流程。

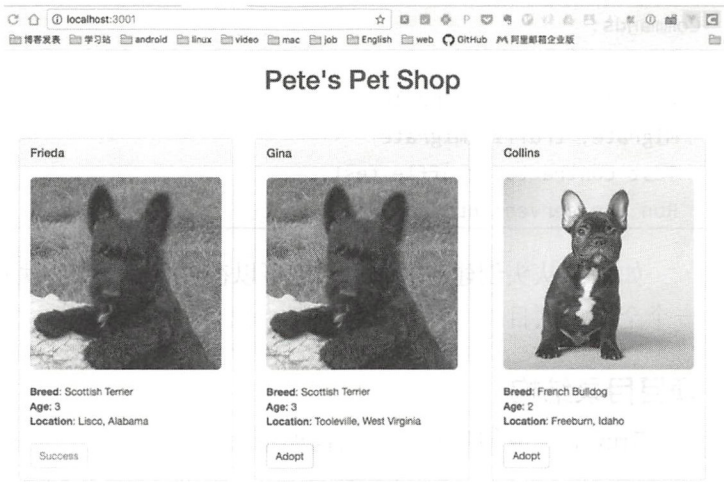
安装 Truffle

使用以下命令安装 Truffle。

```
npm install -g truffle
```

Truffle 使用案例

我们结合案例来看看是如何使用 Truffle 框架编写去中心化应用的。这个应用写的是一个宠物店，在应用中售卖宠物。用区块链记录宠物的领养数据，应用效果如下图所示。



创建项目

首先建立项目目录并进入，代码如下所示。

```
> mkdir pet-shop-tutorial
> cd pet-shop-tutorial
```

使用 Tuffle Unbox 初始化项目。针对不同类型的应用，Truffle 提供了一些 Box。我们创建好了一些示例代码，大家可以通过这个网址（http://truffleframework.com/ boxes/）查看到 Truffle 提供的 Box 情况。

> 这个就像很多 IDE 在新建工程时的引导一样，给我们提供了一些示例代码。

本节介绍的 Pet Shop 应用，已经为我们提供了网站代码，我们只需要编写合约及交互部分即可。通过 truffle unbox pet-shop 来初始化项目，执行成功后输入以下代码。

```
> truffle unbox pet-shop
 Downloading...
 Unpacking...
 Setting up...
 Unbox successful. Sweet!

Commands:

  Compile: truffle compile
  Migrate: truffle migrate
  Test contracts: truffle test
  Run dev server: npm run dev
```

如果想从头创建一个项目，也可以在项目目录下，执行 truffle init 来初始化一个全新的项目。

项目目录结构

Truffle 生成的项目目录结构如下。

- contracts 为智能合约的文件夹，所有的智能合约文件都放置在这里。
- migrations 是用来处理部署（迁移）智能合约的。迁移是用一个额外、特别的合约来保存的。
- test 智能合约测试用例文件夹。
- truffle.js 配置文件。
- src web 源码文件夹。

编写智能合约

接下来，编写智能合约。在 contracts 目录下，添加合约文件 Adoption.sol。

```
pragma solidity ^0.4.17;

contract Adoption {

  address[16] public adopters;  // 保存领养者的地址

    // 领养宠物
  function adopt(uint petId) public returns (uint) {
```

```
    require(petId >= 0 && petId <= 15);  // 确保 id 在数组长度内

    adopters[petId] = msg.sender;        // 保存调用地址
    return petId;
  }

  // 返回领养者
  function getAdopters() public view returns (address[16]) {
    return adopters;
  }

}
```

智能合约很简单，用状态变量 adopters 来保存每个领养者的地址。

智能合约编译

Truffle 集成了一个开发者控制台，在项目目录下运行。

```
> truffle compile
```

输出如下命令。

```
Compiling ./contracts/Adoption.sol...
Writing artifacts to ./build/contracts
```

truffle compile 会把编译好的 ABI 及字节码等信息保存在 build/contracts/Adoption.json 上。

智能合约部署

编译之后，就可以部署到区块链上了。

在 migrations 文件夹下已经有一个 1_initial_migration.js 部署脚本，用来部署 Migrations.sol 合约。Migrations.sol 是 Truffle 生成的一个用来确保不会重复部署的合约，这个合约可以在 contracts 找到。我们来创建一个自己的部署脚本 2_deploy_contracts.js。

```
var Adoption = artifacts.require("Adoption");

module.exports = function(deployer) {
```

```
  deployer.deploy(Adoption);
};
```

同样，和本章介绍的第一个案例一样。在执行部署之前，需要确保有一个区块链环境在运行。使用 Ganache 来启动一个模拟开发链，接下来执行部署命令。

```
> truffle  migrate
```

执行后，有以下类似的输出。

```
Using network 'develop'.

Running migration: 1_initial_migration.js
  Deploying Migrations...
  ... 0x3076b7dac65afc44ec51508bf6f2b6894f833f0f9560ecad2d6d41ed98a4679f
  Migrations: 0x8cdaf0cd259887258bc13a92c0a6da92698644c0
Saving successful migration to network...
  ... 0xd7bc86d31bee32fa3988f1c1eabce403a1b5d570340a3a9cdba53a472ee8c956
Saving artifacts...
Running migration: 2_deploy_contracts.js
  Deploying Adoption...
  ... 0x2c6ab4471c225b5473f2079ee42ca1356007e51d5bb57eb80bfeb406acc35cd4
  Adoption: 0x345ca3e014aaf5dca488057592ee47305d9b3e10
Saving successful migration to network...
  ... 0xf36163615f41ef7ed8f4a8f192149a0bf633fe1a2398ce001bf44c43dc7bdda0
Saving artifacts...
```

智能合约部署好之后，可以看到 Ganache 里区块链状态发生了变化，产生了新区块。

智能合约测试

我们在开发应用的时候，还有很重要的一步要做，那就是进行测试。测试用例可以用 JavaScript 或 Solidity 来编写，这里使用 Solidity。

在 test 目录下新建一个 TestAdoption.sol，编写测试合约代码如下。

```
pragma solidity ^0.4.17;

import "truffle/Assert.sol";   // 引入的断言
import "truffle/DeployedAddresses.sol";  // 用来获取被测试合约的地址
import "../contracts/Adoption.sol";      // 被测试合约

contract TestAdoption {
  Adoption adoption = Adoption(DeployedAddresses.Adoption());

  // 领养测试用例
  function testUserCanAdoptPet() public {
    uint returnedId = adoption.adopt(8);

    uint expected = 8;
    Assert.equal(returnedId, expected, "Adoption of pet ID 8 should be
recorded.");
  }

  // 宠物所有者测试用例
  function testGetAdopterAddressByPetId() public {
    // 期望领养者的地址就是本合约地址，因为交易是由测试合约发起的
    address expected = this;
    address adopter = adoption.adopters(8);
    Assert.equal(adopter, expected, "Owner of pet ID 8 should be recorded.");
  }

    // 测试所有领养者
  function testGetAdopterAddressByPetIdInArray() public {
  // 领养者的地址就是本合约地址
    address expected = this;
    address[16] memory adopters = adoption.getAdopters();
    Assert.equal(adopters[8], expected, "Owner of pet ID 8 should be recorded.");
  }
}
```

Assert.sol 及 DeployedAddresses.sol 是由 Truffle 框架提供的。在 test 目录下不提供 Truffle 目录。

TestAdoption 测试合约同样也很简单，大家直接阅读注释就可以理解。

运行测试用例

在终端中执行。

```
truffle test
```

如果测试通过，则终端输出。

```
Using network 'develop'.

Compiling ./contracts/Adoption.sol...
Compiling ./test/TestAdoption.sol...
Compiling truffle/Assert.sol...
Compiling truffle/DeployedAddresses.sol...


  TestAdoption
    ✓ testUserCanAdoptPet (62ms)
    ✓ testGetAdopterAddressByPetId (53ms)
    ✓ testGetAdopterAddressByPetIdInArray (73ms)


  3 passing (554ms)
```

创建用户接口和智能合约交互

我们已经编写、部署及测试了智能合约。现在我们为合约编写 UI，让 UI 和合约能真正交互起来。如果是使用 truffle unbox pet-shop 初始化的工程，那么已经包含了应用的前端代码。代码在 src/文件夹下。在编辑器中打开 src/js/app.js 可以看到用来管理整个应用的 App 对象。其中 init 函数可以加载宠物信息，并且初始化 Web3。

初始化 Web3

编辑 app.js 修改 initWeb3()函数，删除注释。

```
initWeb3: function() {
   if (typeof web3 !== 'undefined') {
     App.web3Provider = web3.currentProvider;
   } else {
```

```
      // If no injected web3 instance is detected, fall back to Ganache
      App.web3Provider = new Web3.providers.HttpProvider
('http://localhost:7545');
    }
    web3 = new Web3(App.web3Provider);

    return App.initContract();
  }
```

同样，在代码中优先使用 Mist 或 MetaMask 提供的 Provider，如果没有则从本地环境创建一个。

实例化合约

Truffle 会帮我们保存合约部署的信息，所以不用像本章第一个案例那样手动填写合约地址，修改 initContract()代码如下。

```
initContract: function() {
  // 加载 Adoption.json，前面提到过 Adoption.json 保存了合约的 ABI 信息及部署后的地
  // 址信息，它在编译合约的时候生成 ABI，在部署的时候追加地址

  $.getJSON('Adoption.json', function(data) {
    // 用 Adoption.json 数据创建一个可交互的 Truffle Contract 合约实例。
    var AdoptionArtifact = data;
    App.contracts.Adoption = TruffleContract(AdoptionArtifact);

    // Set the provider for our contract
    App.contracts.Adoption.setProvider(App.web3Provider);

    // Use our contract to retrieve and mark the adopted pets
    return App.markAdopted();
  });
  return App.bindEvents();
}
```

处理领养

修改函数 markAdopted()代码。

```
markAdopted: function(adopters, account) {
  var adoptionInstance;

  App.contracts.Adoption.deployed().then(function(instance) {
    adoptionInstance = instance;

    // 调用合约的 getAdopters(), 用 call 读取信息不用消耗 gas
    return adoptionInstance.getAdopters.call();
  }).then(function(adopters) {
    for (i = 0; i < adopters.length; i++) {
      if (adopters[i] !== '0x0000000000000000000000000000000000000000') {
        $('.panel-pet').eq(i).find('button').text('Success').attr('disabled
', true);
      }
    }
  }).catch(function(err) {
    console.log(err.message);
  });
 }
```

修改函数 handleAdopt()代码。

```
handleAdopt: function(event) {
  event.preventDefault();

  var petId = parseInt($(event.target).data('id'));

  var adoptionInstance;

  // 获取用户账号
  web3.eth.getAccounts(function(error, accounts) {
    if (error) {
      console.log(error);
    }

    var account = accounts[0];

    App.contracts.Adoption.deployed().then(function(instance) {
```

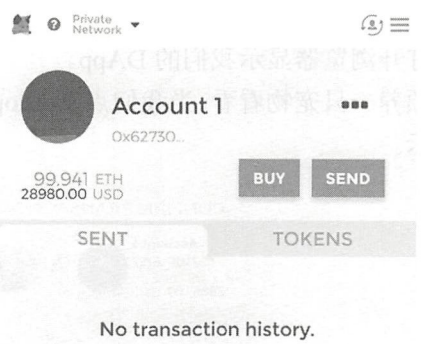
```
        adoptionInstance = instance;

        // 发送交易领养宠物
        return adoptionInstance.adopt(petId, {from: account});
      }).then(function(result) {
        return App.markAdopted();
      }).catch(function(err) {
        console.log(err.message);
      });
    });
  }
```

在浏览器中运行

> 之前安装过 MetaMask 的用户可以直接使用 MetaMask 账号导入功能，用 Ganache 提供的私钥导入对应的账号。

连接开发区块链网络

默认连接的是以太坊主网（左上角显示），选择 Custom RPC，添加一个网络地址 http://127.0.0.1:7545，点击返回后，显示如下。



左上角显示为 Private Network，账号是 Ganache 中默认的第一个账号。

至此 MetaMask 的安装配置已经完成。

安装和配置 lite-server

接下来需要本地的 Web 服务器提供页面服务访问。Truffle Box pet-shop 里

提供了一个 lite-server 可以直接使用，我们看看它是如何工作的。bs-config.json 指示了 lite-server 的工作目录。

```
{
  "server": {
    "baseDir": ["./src", "./build/contracts"]
  }
}
```

./src 是网站文件目录。./build/contracts 是合约输出目录。

与此同时，在 package.json 文件的 scripts 中添加了 dev 命令。

```
"scripts": {
  "dev": "lite-server",
  "test": "echo \"Error: no test specified\" && exit 1"
},
```

当运行 npm run dev 的时候，就会启动 lite-server。

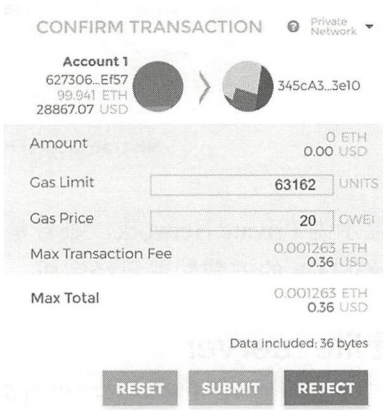
启动服务

启动服务代码，如下所示。

```
> npm run dev
```

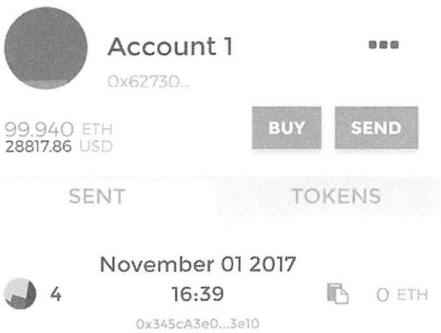
自动打开浏览器显示我们的 DApp。

现在领养一只宠物看看，当我们点击 Adopt 时，MetaMask 会提示确认交易，如下图所示。



点击 SUBMIT 确认后，就可以看到我们已成功领养了一只宠物。

在 MetaMask 中，也可以看到交易的清单，如下图所示。



至此，去中心化式应用完整的开发过程就完成了。

本章小结

本章在编写时，参考了以下文章：

- https://coursetro.com/posts/code/99/Interacting-with-a-Smart-Contract-through-Web3.js-(Tutorial)
- http://truffleframework.com/tutorials/pet-shop
- https://github.com/xilibi2003/dapp-guide-pet-shop

通过本章的介绍，大家可以了解到去中心化应用 DApp 开发的思路及流程。在 DApp 开发中，以太坊的节点充当了像传统应用的后台服务器的一个角色，但是交易请求是由整个网络负责处理的，去中心化应用连接的节点无法干预交易请求的执行，这也是去中心化应用提供可信任的基础，希望本章的案例能给大家开发 DApp 带来启发。

不同于现在市面上大多数书籍从概念、趋势、给社会经济带来的变革等层面介绍区块链技术，本书完完全全是一本面向开发者的技术书籍。本书的一大特点是全面，不但覆盖了以太坊智能合约开发语言Solidity的每一个知识点，而且覆盖了如何基于以太坊进行去中心化应用的开发，包括各个开发工具及框架的使用。建议所有想加入区块链领域的开发者阅读。

—— 西祠胡同创始人、FIBOS创始人　响马

Tiny熊是我所认识的最踏实细心的区块链技术布道者。他可以为读者朋友的每一个程序问题反复斟酌、细心回答。他的博客文章《深入浅出区块链》，让很多程序员成功转型为区块链开发人员。本书延续了《深入浅出区块链》的风格，并且对以太坊的智能合约及DApp开发进行了更加系统、全面的介绍。建议想从事区块链相关工作的开发者及想进入区块链领域的从业者阅读本书。

—— 工信部区块链应用研究院副院长、GEOC（绿色生态）基金会发起人、小牛区块链创始人 赖一诚

以太坊已经慢慢变成区块链的事实标准，大多数开发者接触到区块链首先了解的便是比特币、以太坊、超级账本。本书作为一本以太坊的入门书籍，不仅有理论讲解，还有代码案例解析，是开发者学习以太坊的优选书籍。

—— HiBlock区块链社区发起人 Bob Jiang

自从以太坊成功上线以来，硅基世界的美好未来如此真实地呈现在我们面前。作为程序员，自然深知一个自带价值转移功能的图灵完备编程环境的巨大价值。在人类从碳基世界向硅基世界跃迁的过程中，这是第一步，也是无比重要的一步。任何一个有志于在区块链领域深入研究的研究者，不掌握以太坊智能合约编程是无法再前进一步的。因此，建议大家从阅读这本书开始，打开进入下一个世代的大门。

—— Arena.one创始人　曹晓钢

在Tiny熊的身上，可以看到工程师的气质，一双善于发现问题的眼睛，持续思考、定位问题和探索问题的大脑，以及锲而不舍、求真求实的精神。希望这本书可以成为区块链世界的燎原火种，让更多的工程师在探索的道路上并肩同行，披荆斩棘，早日在价值互联网探索的道路上有所成就。

—— 无退社区创始人、Think技术社区、PHPConChina联合创始人　锅巴GG

以太坊是成熟的区块链开发平台，而区块链开发最重要的就是智能合约开发。熊丽兵是国内最早的区块链开发者之一，他所写的一系列区块链开发相关教程帮助了非常多的开发者。现在他把智能合约开发教程整理成书，可以极大地方便开发者系统地学习智能合约开发。

—— Egretia技术负责人 Dily



博文视点Broadview



@博文视点Broadview

策划编辑：官　杨
责任编辑：牛　勇
封面设计：吴海燕

上架建议：区块链-智能合约

ISBN 978-7-121-34951-5

定价：59.00元